NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

**TOOLS FOR
BINARY DECISION DIAGRAM
ANALYSIS**

by

Kwee Hua Ang

March, 1995

Thesis Advisor: Jon T. Butler

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1995.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE TOOLS FOR BINARY DECISION DIAGRAM ANALYSIS

5. FUNDING NUMBERS

6. AUTHOR(S) Kwee Hua Ang

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT

The Binary Decision Diagram (BDD) is a very useful representation in the design and verification of switching functions. This is due to to its compactness, where size is measured by the number of nodes. In the implementation of logic circuits, connection of sub-functions is by means of pass transistors. The delay time for the interconnections is often larger than the delay of the decision logic. For that reason, crossings are often more expensive than logic. Planar Binary Decision Diagrams are therefore desirable in implementing logic circuits. This paper presents a method for finding a planar Ordered Binary Decision Diagram (OBDD) for threshold functions. The program that implements the algorithm is written in Borland C++. A special case of Fibonacci threshold function having up to 9 variables is analyzed. It is shown that Fibonacci functions having up to 9 variables have planar OBDD. With this program, the characteristics of other threshold functions are developed.

14. SUBJECT TERMS Binary Decision Diagram, Threshold functions, Fibonacci functions.

15. NUMBER OF PAGES 92

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)

Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

TOOLS FOR BINARY DECISION DIAGRAM ANALYSIS

Kwee Hua Ang
Major, Republic of Singapore Airforce
BSEE. FH Furtwangen (W. Germany), 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1995

Author: _____
Kwee Hua Ang

Approved by: _____
Jon T. Butler, Thesis Advisor

_____
David S. Herscovici, Second Reader

_____
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The Binary Decision Diagram (BDD) is a very useful representation in the design and verification of switching functions. This is due to to its compactness, where size is measured by the number of nodes. In the implementation of logic circuits, connection of sub-functions is by means of pass transistors. The delay time for the interconnections is often larger than the delay of the decision logic. For that reason, crossings are often more expensive than logic. Planar Binary Decision Diagrams are therefore desirable in implementing logic circuits. This paper presents a method for finding a planar Ordered Binary Decision Diagram (OBDD) for threshold functions. The program that implements the algorithm is written in Borland C++. A special case of Fibonacci threshold function having up to 9 variables is analyzed. It is shown that Fibonacci functions having up to 9 variables have planar OBDD. With this program, the characteristics of other threshold functions are developed.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Boolean algebra is used widely in computer science and digital system design. Many problems in digital logic design and testing, artificial intelligence and combinatorics can be expressed in a sequence of operations on Boolean functions. However, the classical representation and manipulation of Boolean functions have many shortcomings. A variety of methods have been developed for representing and manipulating Boolean Functions. Those based on classical representations such as truth tables, Karnaugh maps, or canonical sum of products form are quite impractical--every function of n arguments has a representation of size 2^n [2]. A more efficient representation is the Binary Decision Diagram (BDD). It has several advantages. Firstly, most commonly encountered functions have a reasonable representation. For instance, all symmetrical functions are represented by graphs where the number of nodes grows at most at a rate proportional to the square of the number of arguments (n^2). The BDD therefore has a more compact representation. Secondly, the reduced form of BDD is canonical. (i.e., every function has a unique representation), [2].

The binary tree can be explained in the form of a Shannon expansion:

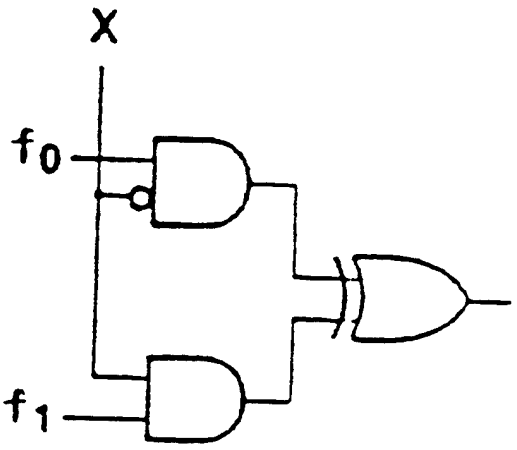


Figure 1. Shannon expansion.

Using the logic circuit of Figure 1 to implement the expansion, we have

$$f = x_1' f_0 .XOR. x_1 f_1$$
$$f0 = x_2' f_{00} .XOR. x_2 f_{01}$$
$$f1 = x_2' f_{10} .XOR. x_2 f_{11}$$

In the same way, we can expand the new subfunctions as follow:

$$f00 = x_3' f_{000} .XOR. x_3 f_{01}$$
$$f01 = x_3' f_{010} .XOR. x_3 f_{011}$$
$$f10 = x_3' f_{100} .XOR. x_3 f_{101}$$
$$f11 = x_3' f_{110} .XOR. x_3 f_{111}$$

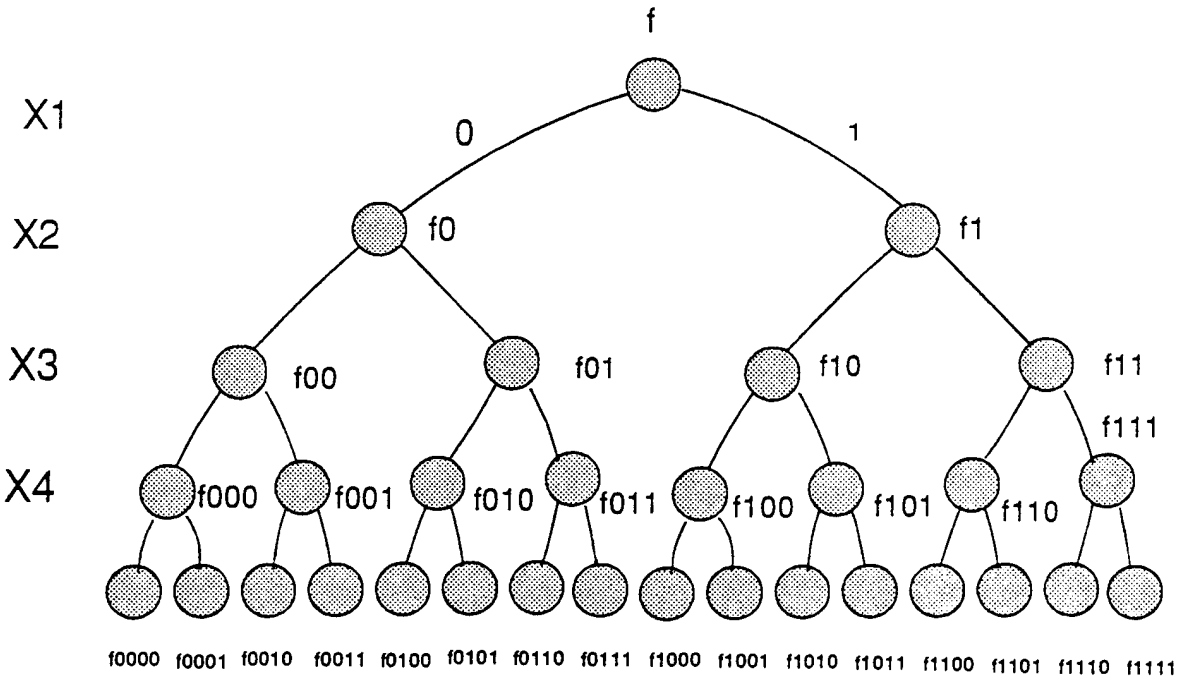


Figure 2. Representation of logic function using Shannon expansion.

If we replace each node by a 2 input multiplexer, we have the following network:

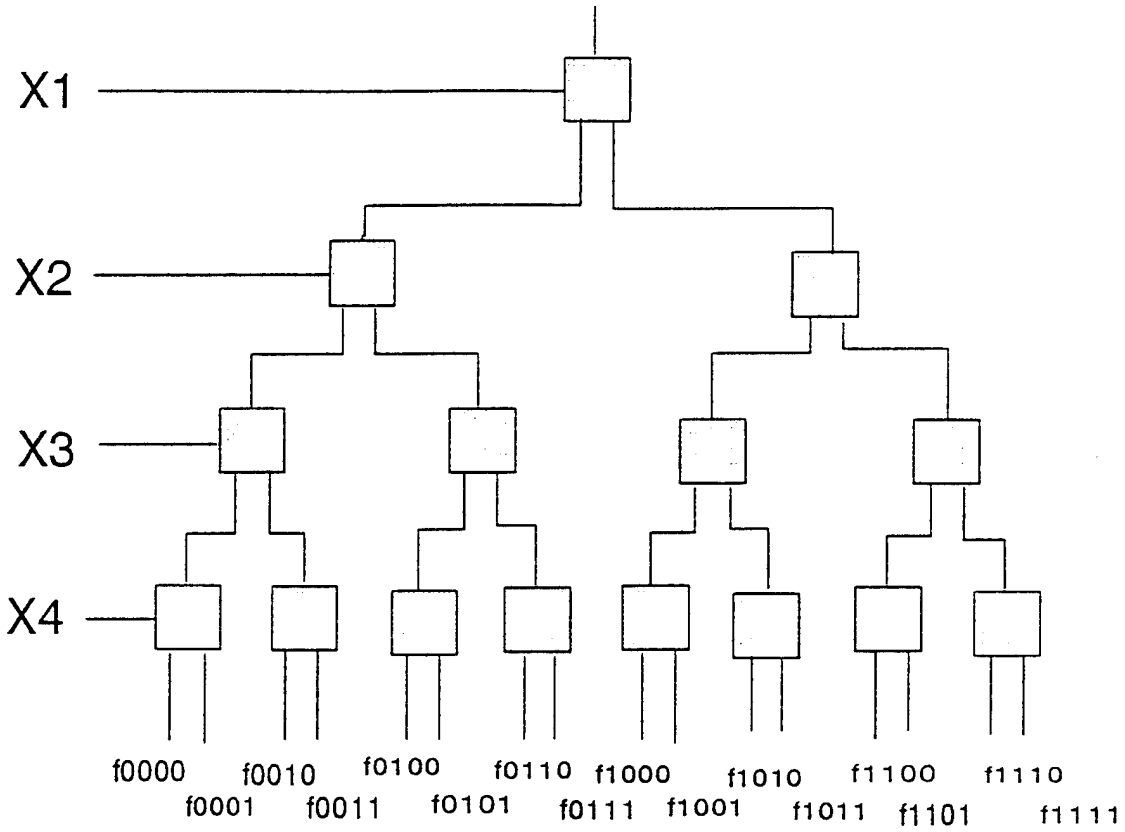


Figure 3. Realization of logic function using multiplexers.

A BDD is a way to represent a given function using a binary tree. A binary tree has a root node which is unique in that it is not a child node of any other node. The root node of a BDD represents the given function $f(x_1, x_2, \ldots x_n)$. The left child of the root represents the subfunction $f(0, x_2, x_3 \ldots x_n)$ and the right child $f(1, x_2, x_3 \ldots x_n)$. Similarly, the grandchildren, great-grandchildren etc. represent the subfunctions associated with x_2 and x_3 and so on, until all variables are assigned. The leaf nodes, which have no children are assigned with constant 0 and 1.

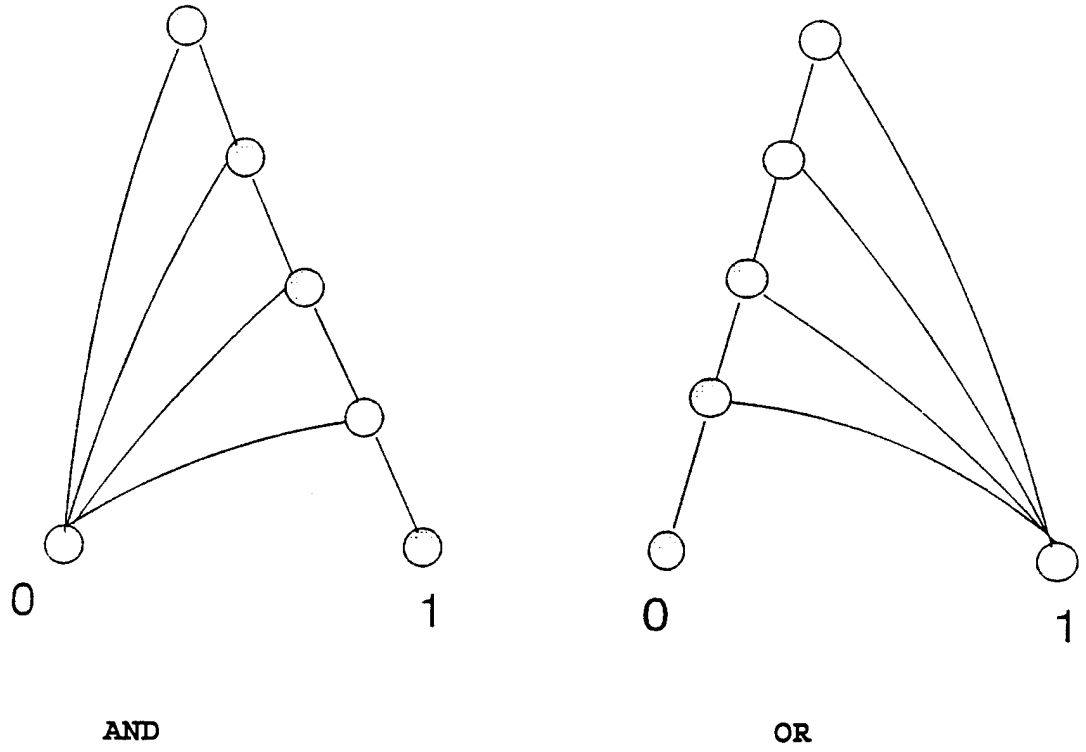
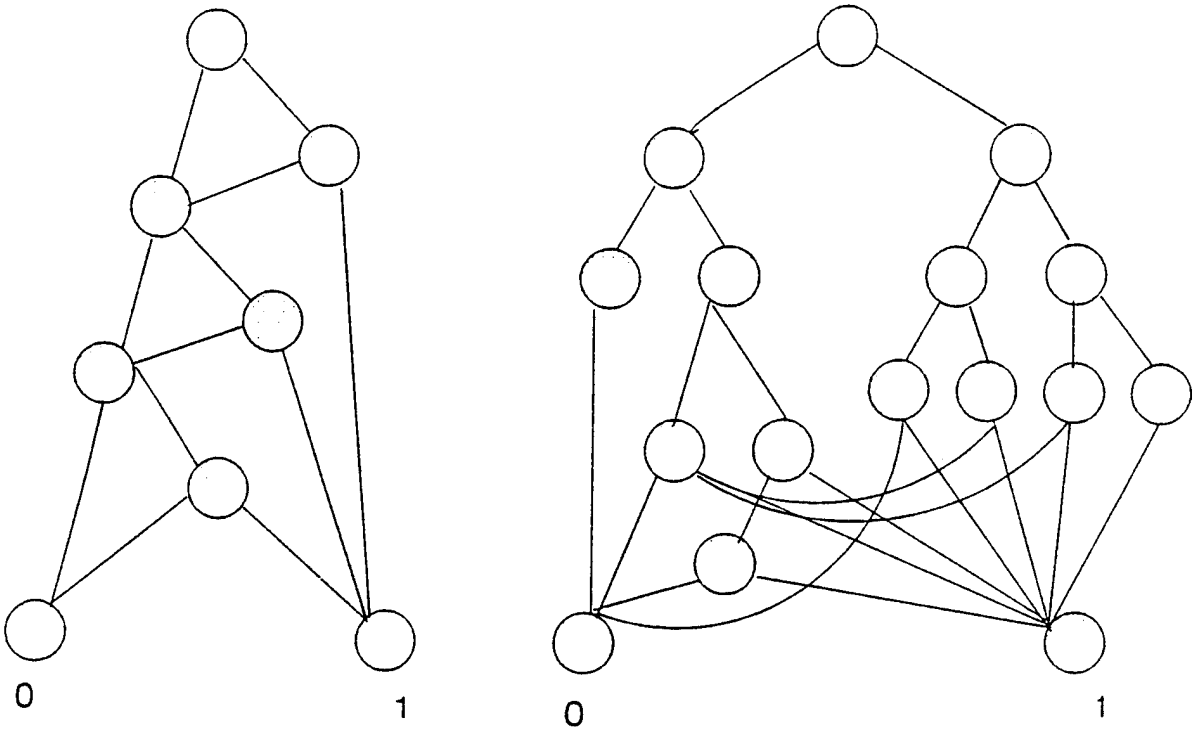


Figure 4. The BDD for a AND and OR function.

Figure 4 shows examples of the BDD for the AND and the OR function. In this case, the ordering of the variables makes no difference in the BDD structure. Ref [2] showed an interesting example of a BDD which is highly dependent on ordering of the variables, (See Figure 5) The functions $x_1x_2 + x_3x_4 + x_5x_6$ and $x_1x_4 + x_2x_5 + x_3x_6$ differ from each other only by a permutation of variables, yet the first BDD has 8 nodes while the second has 16 nodes. From this example, one can see that a poor choice of ordering can have very undesirable results.



$x_1x_2 + x_3x_4 + x_5x_6$ $x_1x_4 + x_2x_5 + x_3x_6$

Figure 5. Example of argument ordering dependence.

Besides the difference in complexity in terms on number of nodes, the functions $x_1x_4 + x_2x_5 + x_3x_6$ has crossings, while the other does not. This shows that BDDs without crossing can be found by varying the order of the variable, if they exists.

In the LSI implementation of a network, crossings are expensive because they require additional channels and increase delay. In submicron LSI, the delay in the interconnections are comparable to the delay in the logic elements. For a complex network, the delay caused by crossings can be unacceptable. It is therefore understandable that one of the important goals in circuit design involving BDD's is to eliminate, or at least reduce crossings in the network.

Sasao & Butler [4] define a restricted planar BDD:

Definition 1: A function has a restricted planar BDD if there exists a BDD without crossings, where the 1 edge emerges to the right of the node and 0 edge emerges to the left of the node; and the constant 1 is in the right of the constant 0.

This definition will be used to describe a planar BDD in this text. Until recently, only BDDs up to 5 variables have been shown to have no crossing [4]. The progress in this field is slow because manual enumerations of BDD are very tedious. To facilitate further research related to BDDs, the process of drawing BDD, searching for those without crossings, and producing their orderings, counting of their nodes etc. must be computerized.

The development of such a tool is one of the main aims of this thesis. The application of this tool is demonstrated by characterizing a class of functions. The following definitions in accordance with [1] are used in the subsequent discussion:

Definition 2: A *switching function* f(x1,x2, ...xn) is a mapping of $f:B^n \rightarrow B$, where B = {0,1}.

Definition 3: A *threshold function* f(x1,x2, ...xn), has the property that f = 1 iff $w_n x_n + w_{n-1} x_{n-1} + \ldots + w_1 x_1 \geq T$, where T and w_i are integers and the logic values, 0 and 1 of x_i are viewed as integers.

The value of $w_n x_n + w_{n-1} x_{n-1} + \ldots + w_1 x_1$, for some assignment of values x1,x2, ...and xn is called the weighted sum. A threshold function is completely specified by a weight-threshold vector, $(w_n, w_{n-1}, \ldots, w_1 ; T)$. AND and OR are special examples of threshold functions. Since any switching function can be realized by a combination of AND, OR and inverter, any switching function can be realized by a network of threshold

functions. In threshold logic, in general, each logic gate represents more than AND,OR or NOT does in conventional switching theory. Accordingly, the number of gate that realize a given function is often less than the number of gates required by the AND,OR, NOT or other conventional gates.

II. REQUIREMENTS OF THE ANALYSIS TOOLS

This thesis describes a tool to analyze BDD's for threshold functions. It can be extended to evaluate other functions. The tool is designed with the following considerations in mind:

1. Inputs

The parameters describing the threshold functions such as the vector $(w_n, w_{n-1}, \ldots, w_2, w_1; T)$ will be requested once the program is started. Several options are provided to tailor the analysis to certain requirements. Since some analysis requires the threshold to be varied, the option to vary the threshold from some minimal value to a maximum value is also provided. If the aim is to search for planar BDDs, the option for producing various ordering of the variables in lexicographical order is available. The option for drawing the BDD and printing in a text file can be activated to help in visual analysis of BDD's. However, for certain BDD structures, when searching a threshold function of many variables, the number of planar BDDs can be very large. A preview of which BDD is planar can be made by requesting a listing of planar BDDs first.

2. Processing

The program is designed to evaluate a threshold function and provide the result in the form of a truth table for further BDD analysis. For other types of functions, the routine to convert the particular function to a truth table must be written. The truth table can be a large table having 2^n entries, where n is the number of variables. Truth tables are useful for checking or interfacing with other programs. It is also used in the generation of the initial BDD.

The program simplifies the initial BDD to a form called reduced ordered BDD. To allow drawing of the BDD, the reduced BDD is laid out in the form of a binary tree.

At present, the program is able to analyze BDDs of up to 9 variables. If threshold functions of more variables are needed, the program can be expanded. The program is designed to detect any crossings in the BDD. This feature would help in searching for planar BDDs in any type of function, and is useful for the design of fast logic circuit. An important output is the number of nodes of a reduced order BDD. This feature is useful since the compactness of a BDD is given by the number of nodes in the BDD.

The program produces all the possible orderings of variables to search for planar BDD, if it exists at all. As discussed before, some orderings yield planar BDDs.

3. Output

The program provides the following output for BDD analysis:

- Layout for drawing of the BDD
- Number of nodes for each BDD
- List of BDD's without crossings, showing the ordering for each BDD

III. ALGORITHM & IMPLEMENTATION IN C++

The algorithm to enumerate binary decision diagrams and analyzing them is described below:

1. Generate Lexicographical Ordering

This module is required for the enumeration of BDDs. It produces all orderings of the variables. If two variables have the same weight, only the first ordering is used; all subsequent orderings of variables corresponding to this same arrangement of weights are ignored. For example, with weight-threshold vector (3,2,1,1; 5), ordering $x_4x_3x_2x_1$ is ignored, and only ordering $x_4x_3x_1x_2$ is used. The program run time can be very long as the number of variables increases since there are n! threshold functions to enumerate.

2. Enumeration of Threshold Function

An important part of the program is to generate a threshold function with weight-threshold vector $(w_n,\ldots,w_2,w_1;T)$ which is stored in an array to be used for binary tree generation at a later stage.

3. Generation of a Binary Tree

A binary tree is generated with $2^{(n+1)}$ nodes. Each of the nodes of the binary tree has the following properties:

- **Index:** Each node is assigned an index that increases from top to bottom and from left to right. This identification is used later to check whether a node can be merged. Figure 6 shows an example of the indexing of the nodes for a binary tree that describes a 4 variables function. For every node of index i, its left branch has an index of *2i*, and its right branch has an index of *2i+1*.

- **Pointer to left branch:** Pointer that point to the left node immediately following that node concerned.

- **Pointer to right branch:** Pointer that point to the right node immediately following that node concerned.

- **Data:** A data value, DATA, associated with all nodes other than leaf nodes (those with index from 2^n to $2^{(n+1)}$) is initialized with the character "*". In the

subsequent steps, other values such as 0,1 or an alphabet "a" ..."j" will be assigned to indicate the size of the subfunction that has been merged. A leaf node corresponds to some assignment of values to all variables. Each such node is labeled by the value of the function for that assignment.

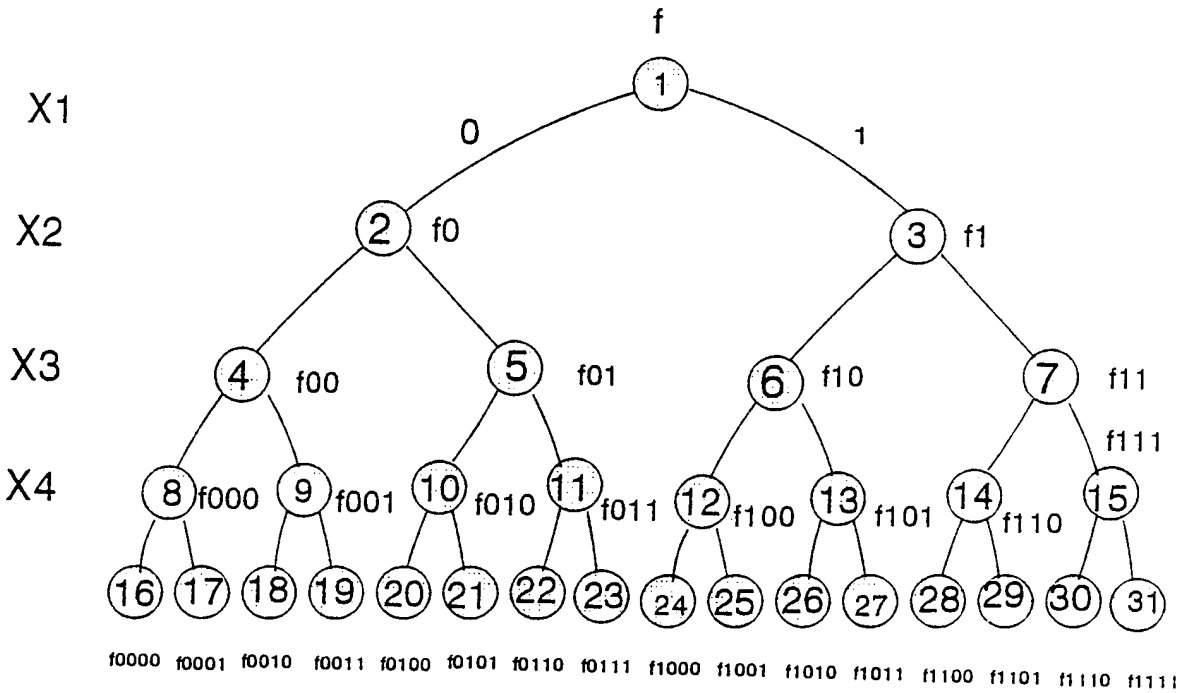


Figure 6. Indexing of Binary tree.

4. Simplification of BDD

Figure 6 is also an example of a full binary decision diagram of a function. Such a BDD has many redundant nodes and can be simplified. The algorithm to do this is as follow:

a. If the value of a subfunction, which is stored in DATA, is a constant 0 or 1, terminate that branch. The DATA of the nodes of the binary tree that proceed after the constant 1 or 0 will be filled with blanks to show that the corresponding branches are terminated.

b. If the current subfunction f_i is the same as one already generated f_j , move the current branch f_i (f_j) over to the other branch f_j. For example, if f01 is the same as f10, move f10 branch over thus merging the 2 nodes. The BDD resulting from the above simplification is called a *Reduced Order Binary Decision Diagram (ROBDD)* [3]. Letters "a" ... "j" are used to indicate the size of the subfunction merged. If the merging is done for a subfunction with only two children and no grandchildren, then the merged subfunction will be assigned a character "a". Similarly, for nodes with two children and 4 grandchildren will be assigned the character "b" if it is a candidate for merging. etc. Such a scheme is implemented for nodes that have up to 2^9 offsprings. See the example in table 1. The merging of the subfunctions is done from "bottom up", which means that comparison of subfunctions are done first for subfunctions with 3 nodes. All subfunctions with 3 nodes are compared and merged if found to be identical. A merged subfunctions with three nodes will be assigned with character "a". After that the next level of subfunctions with 7 nodes will be compared. The process goes on until all subfunctions are evaluated for their potential to merge. The size of the largest subfunction compared is 2^n+1.

Number of nodes in subfunction merged	3	7	15	31	63	127	255
Descriptor	a	b	c	d	e	f	g

Table 1: Example of descriptor for subfunction.

c. If two subfunctions of one node n are the same, i.e. the node's left branch n_0 and the right branch n_1 are the same, then extend the branch to n down to the n_0 and n_1 eliminating n. (i.e., do not use a multiplexer).

5. Counting Nodes

The number of nodes is a useful measure of the compactness of a BDD. C++ has a recursive binary tree traversal function that is very convenient for this purpose. A valid node has the characteristics of DATA = "*" with two children branches.

6. Check Crossings

Inorder traversal is used in the checking of crossing. The steps for inorder traversal are :

1. Traverse left subtree using an inorder traversal.
2. Visit the root node.
3. Traverse right subtree using an inorder traversal.

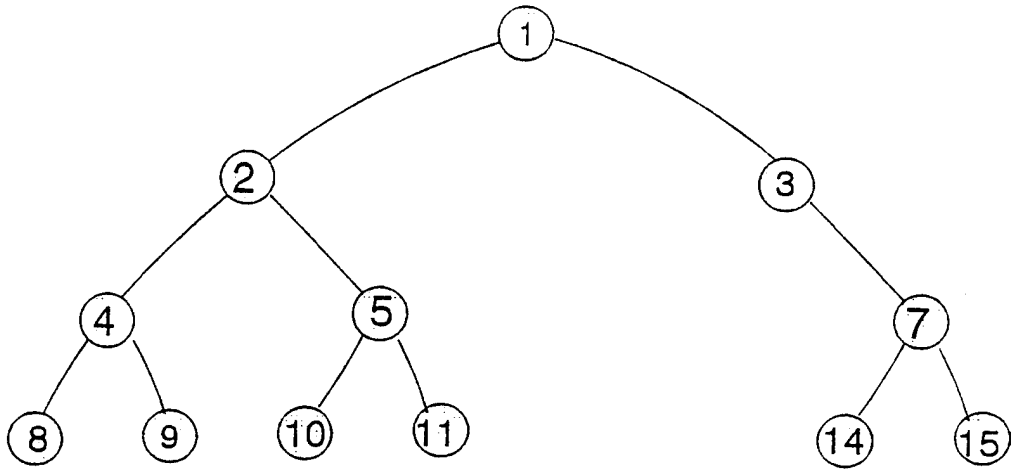


Figure 7. Example of inorder traversal. The order of traversal for the above Binary tree is 8-4-9-2-10-5-11-1-3-14-7-15.

Using the recursive C++ function "inorder traversal", the DATA elements of each node of the Reduced Ordered Binary Diagram are checked consecutively for a transition of "0" to "1" . Not all nodes have a 0 or 1. When a 0 or 1 is encountered, it is recorded to check for 0-1 transition. For a reduced order planar BDD, there should only be one such transition during the complete traversal of the binary tree. i.e., All nodes preceding the last "0" must be "0", "*" or any symbol representing a merged subfunction. Likewise, any node after the first occurrence of a "1" must be either "1" or "*" or any of the symbol representing a merged subfunction. A BDD with more than one 0-1 transition in a inorder traversal of the reduced binary tree will exhibit crossings, since all leaves with "0" can only be joined to each other with crossing, and likewise those leaves with "1". Figures 8

through 11 show how this algorithm works with threshold function with weight-threshold vector (5,3,2,1,1; 7).

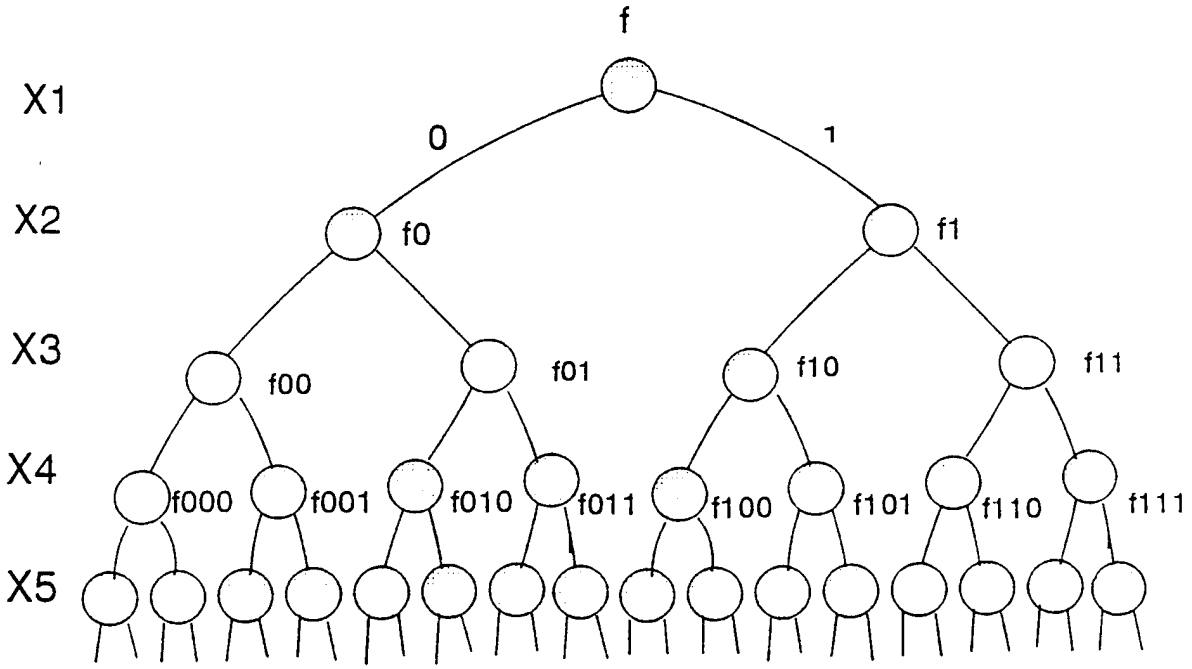


Figure 8. (Step 1.) Enumeration of the threshold function with weight-threshold vector (5,3,2,1,1;7).

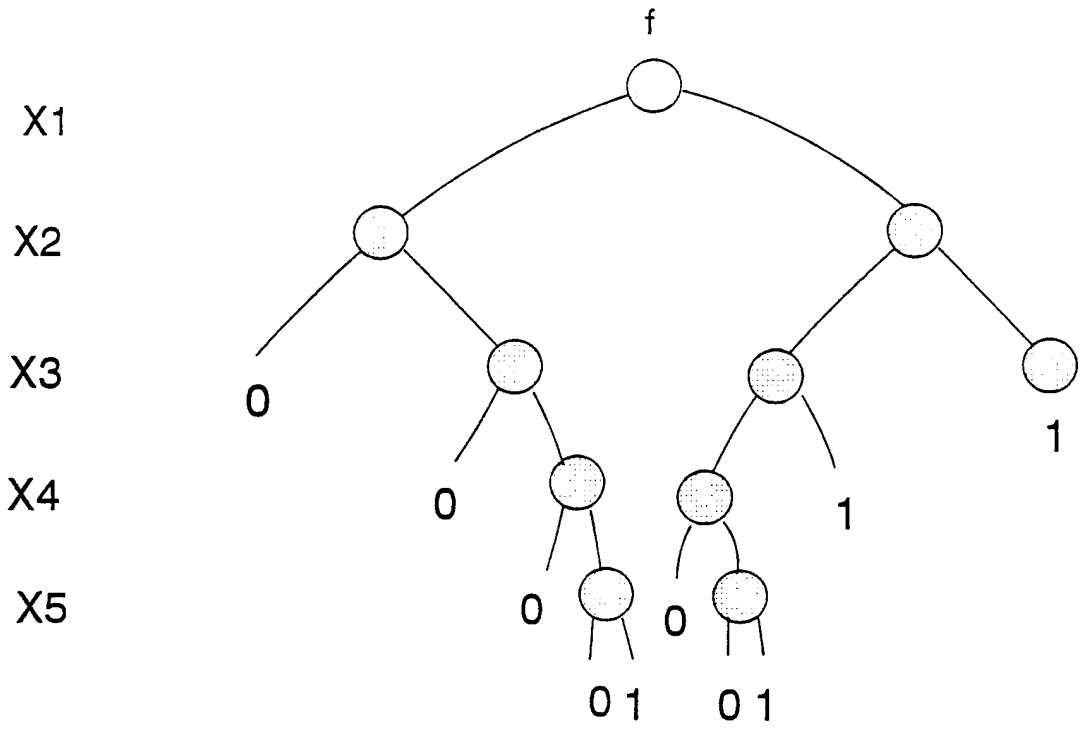


Figure 9. (Step 2.) Simplification of BDD.

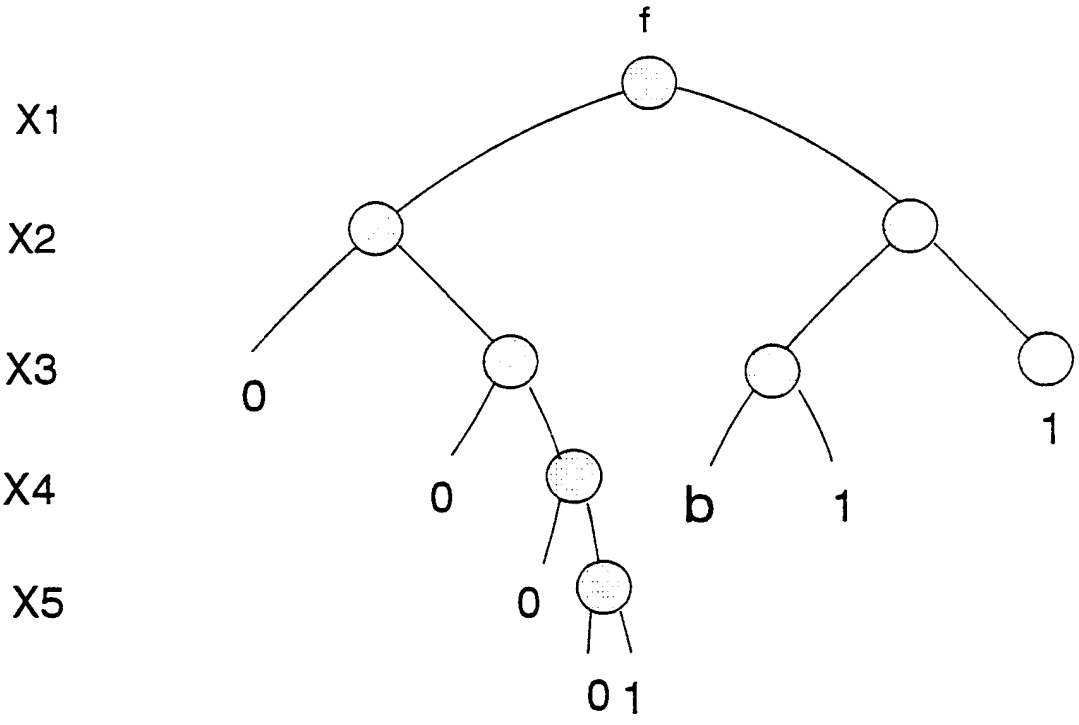


Figure 10. (Step 3.) Printing of simplified BDD layout.

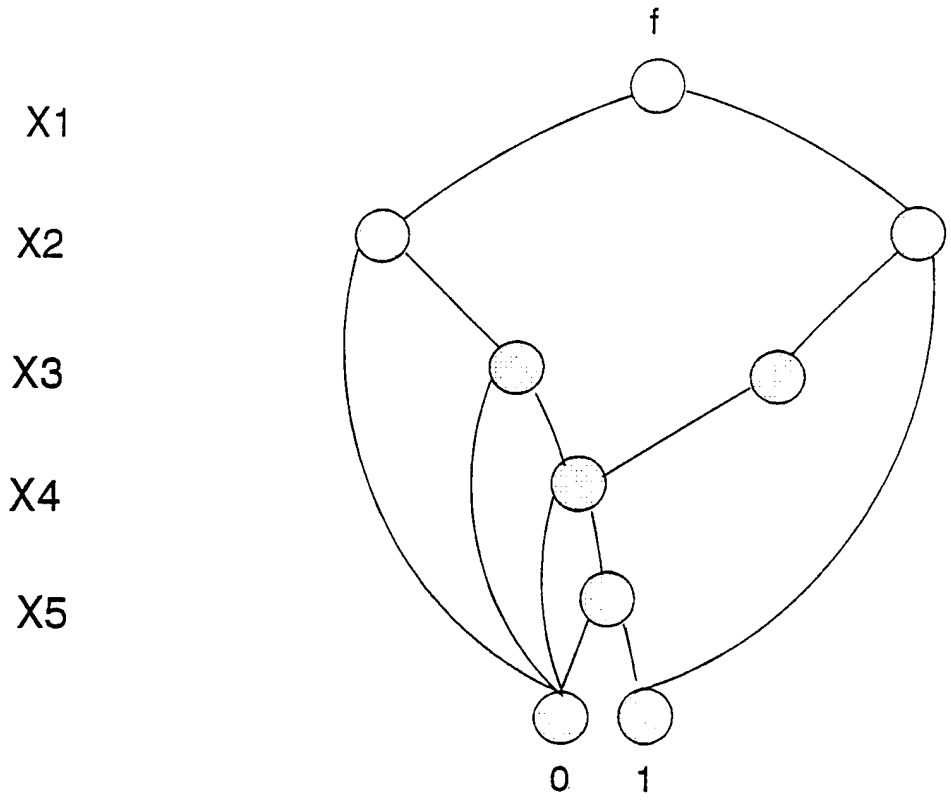


Figure 11. (Step 4.) Completion of Planar BDD

The current implementation of the algorithm is able to analyze threshold functions of up to 9 variables. Figures 12-15. are examples of 6,7,8 and 9 variable Fibonacci threshold functions represented in the form of planar BDD.

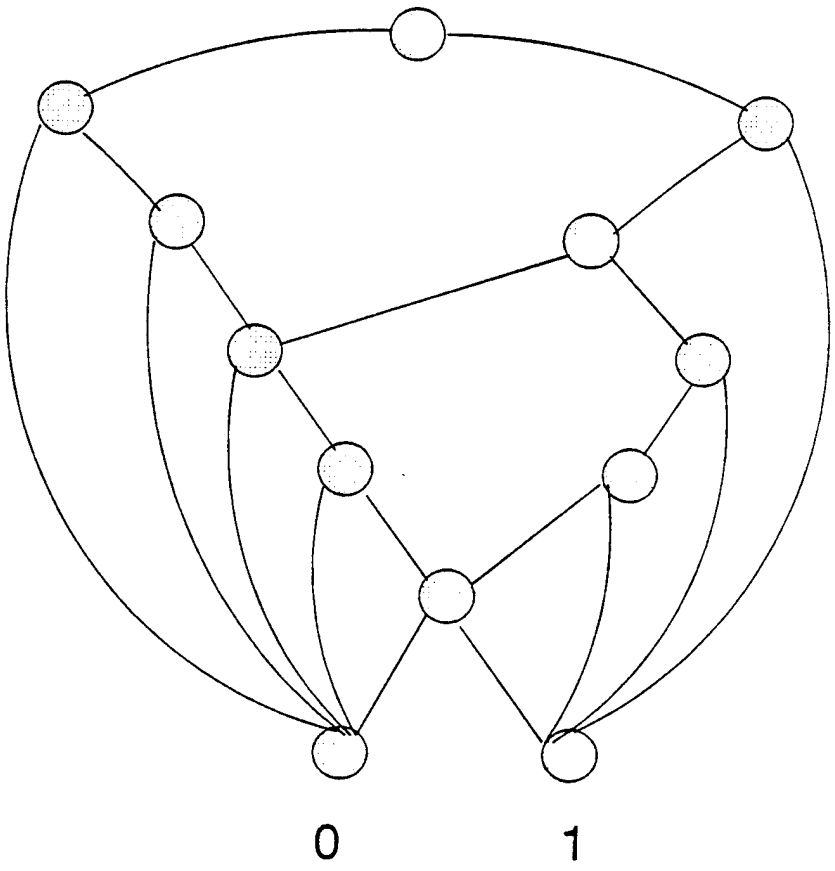


Figure 12. Example of a planar BDD for function with weight-threshold vector of (8,5,3,2,1,1; 12).

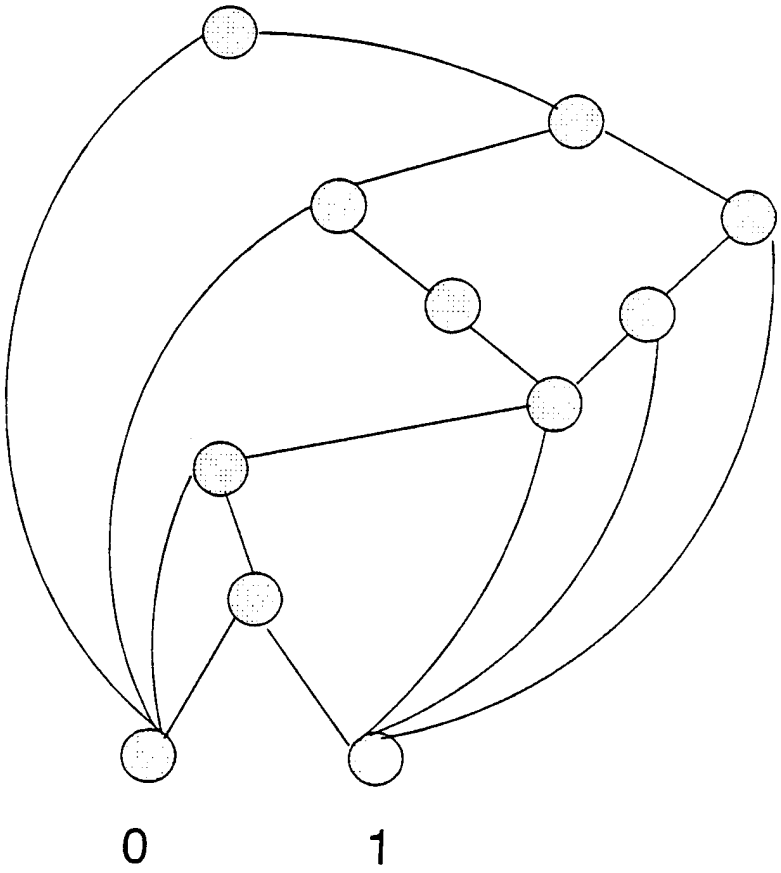


Figure 13. Example of a planar BDD for threshold function with weight-threshold vector of (13,8,5,3,2,1,1; 23).

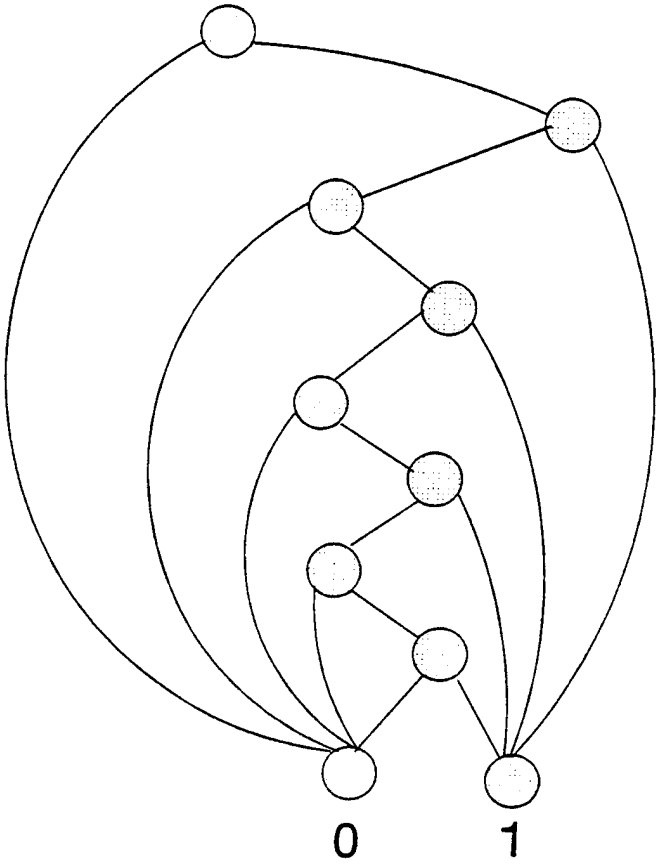


Figure 14. Planar BDD for function with weight-threshold vector (21,13,8,5,3,2,1,1; 34).

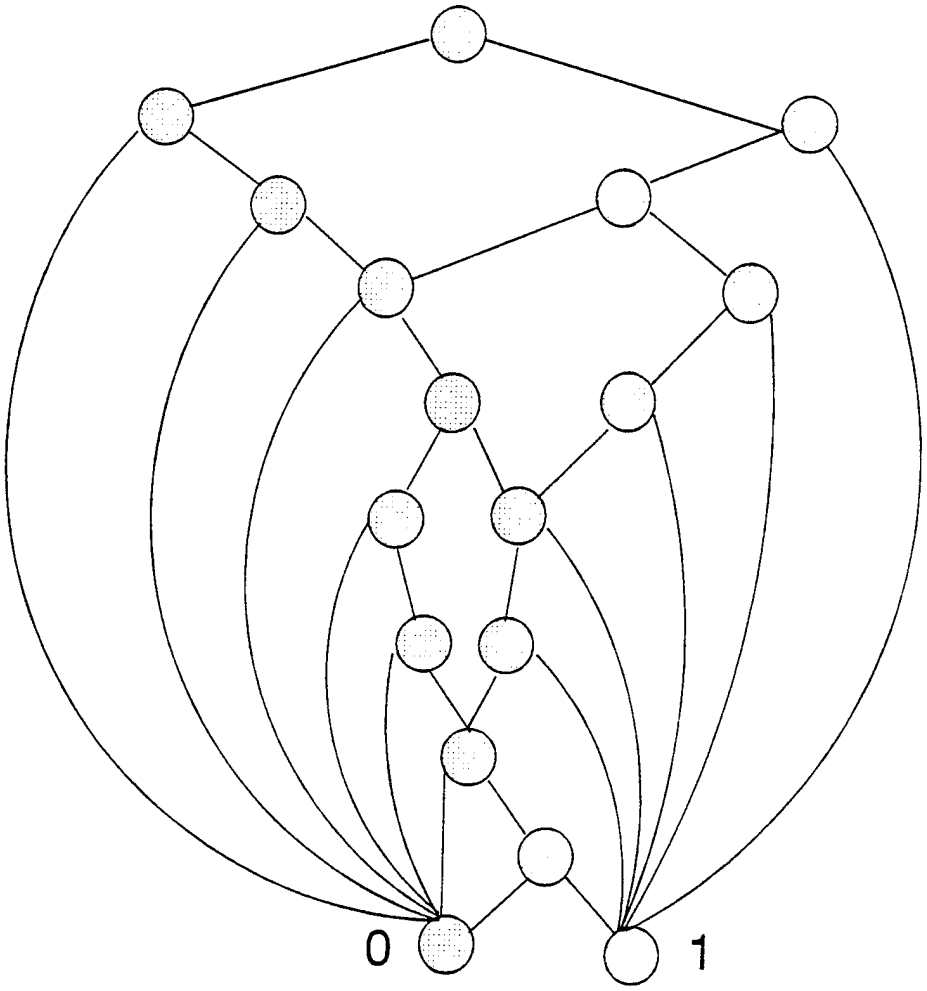


Figure 15. Planar BDD for Fibonacci threshold function with weight-threshold vector (34,21,13,8,5,3,2,1,1; 50).

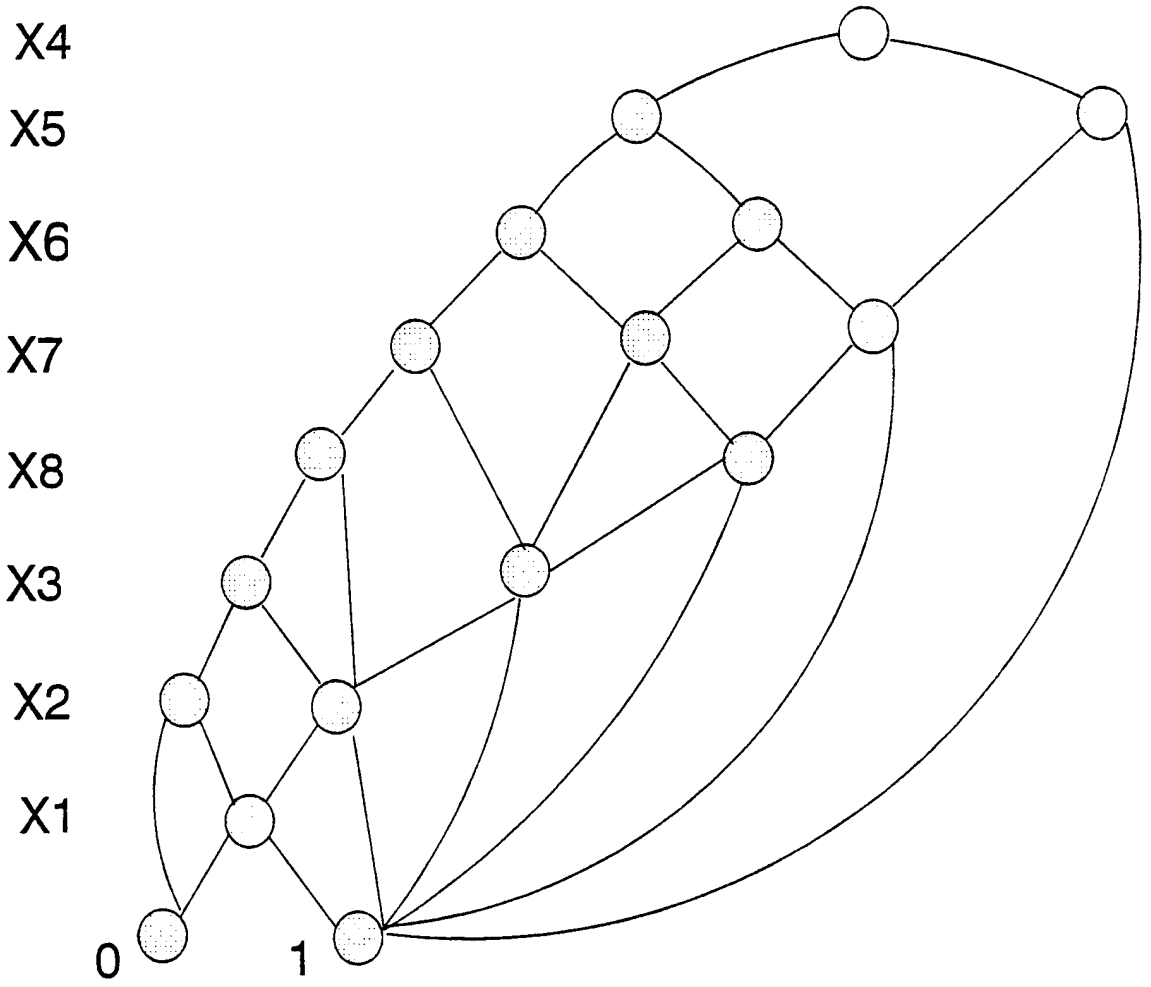


Figure 16. Planar BDD for threshold function with weight-threshold vector (6,7,5,4,4,3,3,2 ; 8).

7. Output

There are two main output forms:

a. **Binary Decision Diagram:** The nodes of the planar BDD is laid out in the form of a binary tree in ASCII text. For a large tree (i.e. of 7 or more variable threshold function), the tree should be viewed or printed out in compressed font due to the limited space. The nodes can be joined together in the same way as a binary tree. The merged subfunctions are represented by alphabets a,b,....j. 'a' corresponds to the subfunction with two leaf nodes, b correspond to a subfunction with two children nodes, each with two leaf nodes etc. These nodes marked by a letter (a,b,...,j) should be joined to the equivalent subfunction that it represent on the left of the binary tree. All nodes with leaf nodes of '0' and '1' should be joined to sink nodes '0' and '1' at the lowest layer of the binary tree.

b. **Listing:** The listing of the planar BDDs without crossing are printed to a ASCII text file in the same directory as the program. Other characterization, such as the number of nodes, ordering etc. are also listed.

IV. ANALYSIS

A. PLANAR BDD FOR CLASSICAL THRESHOLD FUNCTIONS

Muroga [6] defines **canonical function** as a function of n variables with $w_1 > w_2 > \ldots > w_n$. He lists all canonical positive threshold functions with five or fewer variables.

For six variables, only canonical positive self dual functions are shown. A dual of the function f(x) is defined as

$$f^d(x) = f'(x')$$

where f' is the complement of f and $x' = (x_1', x_2', \ldots x_n')$ and A function is self dual if

$$f^d(x) = f(x)$$

x_n' is the complement of x_n.

Using the methodology described in this thesis, the ordering that produces planar BDDs for each of the listed threshold functions is found. The orderings that correspond to these planar BDD are listed in Appendix (A). From this, we can state

Theorem 1 : All threshold functions of up to five variables and all six variables canonical positive self dual threshold functions have a planar BDD.

Based on the ordering suggested by the program written, the BDD can be either derived manually or by means of the BDD program whose source code is listed in APPENDIX (C), for all threshold functions listed in the APPENDIX (A). This table greatly simplifies the design of efficient logic network based on planar BDD for common threshold functions. Giving the number of nodes that corresponds to each planar BDD helps in further optimizing the logic design, by indicating which ordering has the minimal number of nodes.

Theorem 2. All threshold functions of up to five variables and all six variables canonical positive self dual threshold function have a planar BDD with the minimum number of nodes.

This shows that planar BDDs are amongst the most efficient representations of the threshold function with a small number of variables. This is shown to be the case for threshold functions up to 8 variables.

The percentage of planar BDD decreases rapidly as the number of arguments increases, which means that planar BDD are extremely rare as the number of arguments increase. The orderings for planar BDD cannot be easily derived heuristically for threshold functions of many variables. There is often more than one minimal BDD. The following table shows the frequency of planar BDD versus all possible BDD's:

No. of variables	1	2	3	4	5	6
Total no. of BDD's	1	2	7	92	2578	1310
Average no. of BDD's that are planar	1	2	2.3	2.7	4.8	5.4
Average no. of BDD's that are planar and minimal	1	2	1.6	2.5	3.86	4.08
Percentage planar BDD (%)	100	100	71.4	44.5	16.1	4.9

Table 2 : Planar BDD is very rare as the number of variables increases. Column 6 refers to 6 variable canonical positive self dual threshold functions.

The **total number of permutations** refers to the total number of permutations on n variables that produce unique functions, for all n-variables threshold functions listed in Appendix A. The **average number of permutations that are planar** is found by dividing the total number of planar BDDs by the number of threshold functions. When such BDD has minimum number of nodes, it is classified as **planar and minimal.**

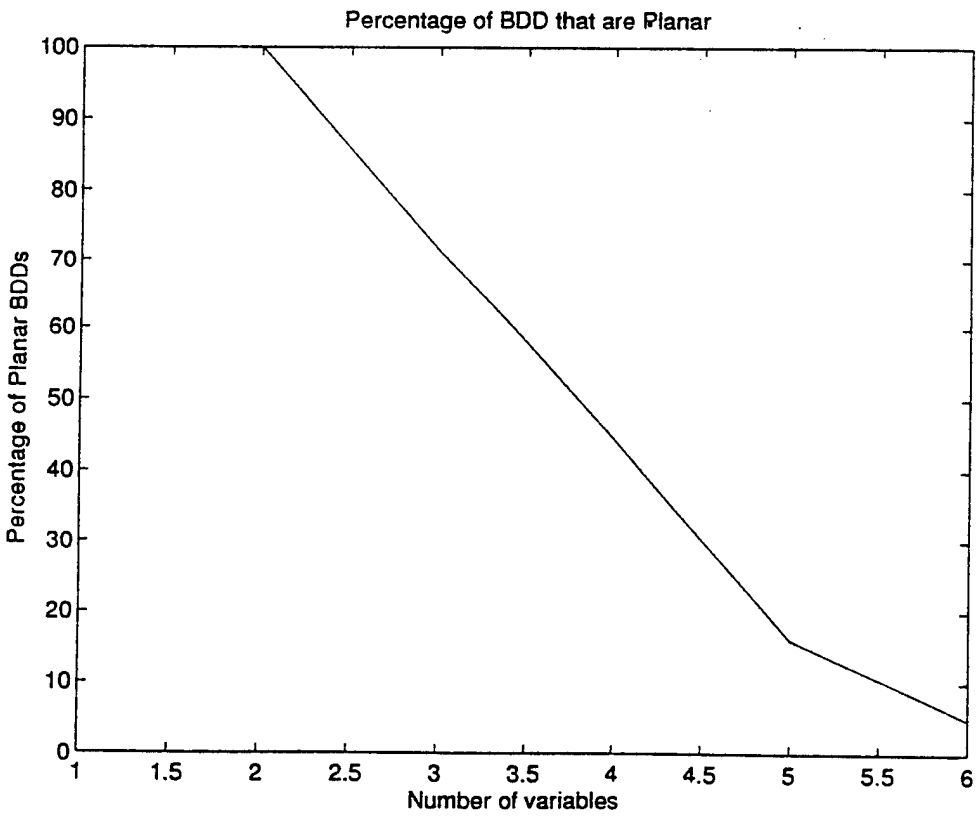


Figure 17. Percentage of planar BDD in all unique permutations.

Figure 17 is a graph of percentage of permutations that yields planar BDD versus the number of variables. Planar BDDs for threshold functions of more than 8 variables are very rare. Although exhaustive search planar BDDs for all threshold function with 8 variables is not done, search on a small sample of 8 variable BDD, such as the one shown in Figure 16 shows that percentage of planar BDDs approaches zero.

B. FIBONACCI FUNCTIONS

Definition 4: A Fibonacci number F_i is specified by a recursive relationship: $F_i = F_{i-1} + F_{i-2}$, with $F_1=F_2=1$.

Definition 5: A *Fibonacci function* is a threshold function with weight-threshold vector $(F_n,F_{n-1},....F_2,F_1;T)$, where F_i is the i-th Fibonacci number and $0 < T < F_{n+2}$.

A *Fibonacci function* is a threshold function with weight-threshold vector $(F_n,F_{n-1},....F_2,F_1;T)$, where F_i is the i-th Fibonacci number and $0 < T < F_{n+2}$. The following have been observed for Fibonacci threshold functions of up to 9 variables.

Theorem 3. All Fibonacci threshold functions with thresholds of one to the largest threshold and with an ascending ordering of variable have planar BDD.s

This is also stated in [1]. The largest threshold is the same as the largest weighted sum, $1+1+2+3+....+F_n=F_{n+2}$. Table 1 in Appendix B shows the number of nodes of Fibonacci functions for threshold of 1 to the largest threshold.

Theorem 4. Amongst the orderings that yield the smallest number of nodes are (1.) descending and (2.) ascending order of weight.

When the threshold is 1, the ordering does not matter at all, since this function corresponds to a OR function. When the threshold is at its maximum value, the planar BDD

correspond to a flipped image of the BDD with threshold of 1. It can be seen from the listing in Appendix B that these always correspond to the minimal BDD.

Theorem 5. The compactness profile of the planar Fibonacci BDD's with thresholds from 1 to its largest threshold is symmetrical.

This symmetry can be seen clearly graphically as shown in Appendix B. Figure 18 shows an example of Fibonacci functions of 9 variables, with the plot of the number of nodes versus the threshold value of 1 to their maximum. The distance from one minimum to the next one in the compactness profile exhibits an interesting characteristic. i.e.,

$$\frac{\text{Distance of } i^{th} \text{ trough to } (i+1)^{th} \text{ trough}}{\text{Distance of } (i+1)^{th} \text{ trough to } (i+2)^{th} \text{ trough}} \sim \text{Golden ratio}$$

with Golden ratio $= \lim_{n \to \infty} \frac{F_n}{F_{n-1}} = 1.618$.

where F_n is a Fibonacci number .

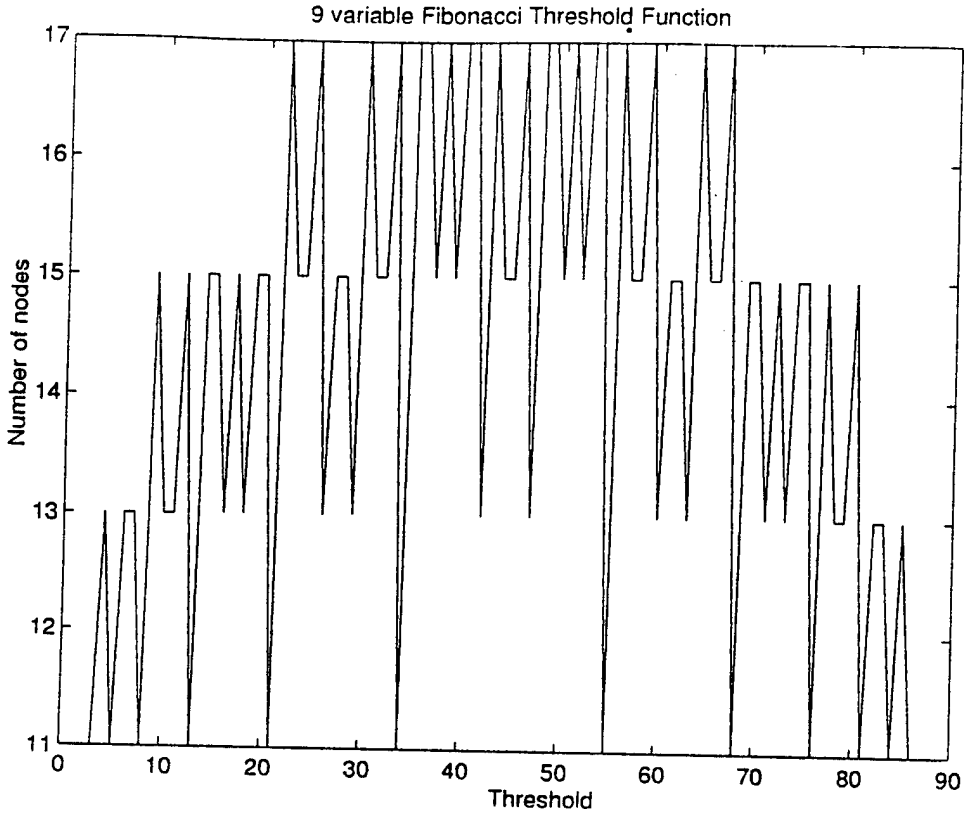


Figure 18. Example of Symmetry of Fibonacci Functions with weight-threshold vector of (34,21,13,8,5,3,2,1,1; T) with thresholds (T) of 1 to 54.

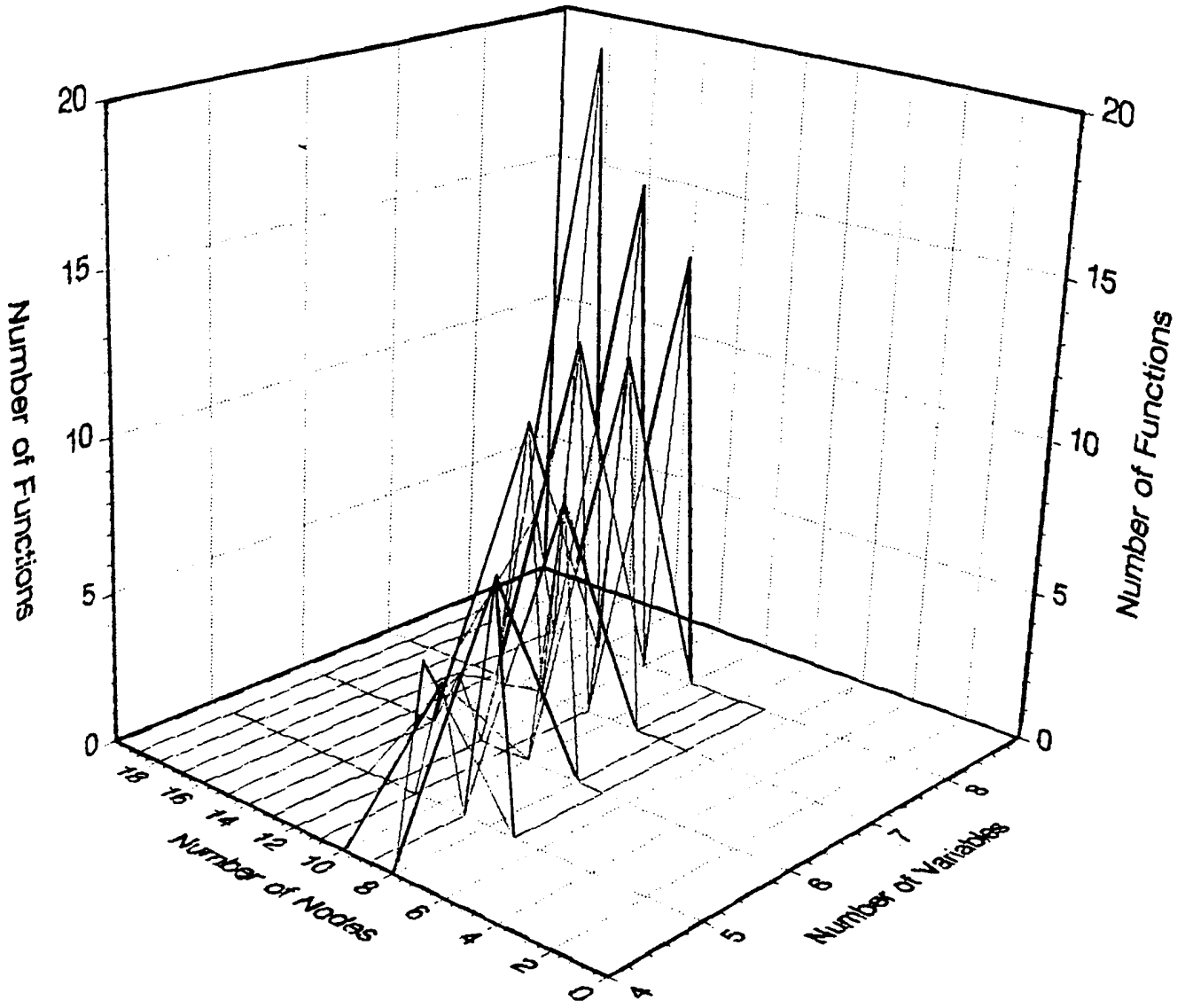


Figure 19. Distribution of Fibonacci function by Nodes and Variables.

Figure 19 shows the distribution of nodes in the BDD's of Fibonacci functions as enumerated by the BDD program. The number of variables and the number of nodes in the BDD's are plotted horizontally, while the number of Fibonacci functions is plotted vertically.

V. CONCLUSION

An effective method of finding planar BDDs for threshold functions is developed. The algorithm is implemented for threshold functions of having up to 9 variables. The source code in Borland C++ is shown in Appendix C. It is demonstrated that Fibonacci threshold functions having up to 9 variables have planar BDDs. The Fibonacci threshold function is also characterized using the algorithm. It is also found that all threshold functions of having up to 6 variables listed in [6] have planar BDDs. The ordering that produce planar BDDs are catalogued in Appendix A for easy reference.

Follow up research in this area may produce an algorithm to convert a BDD to its sum-of-product expression. This would facilitate research in more effective representation of switching functions. Currently, the program does not accept other functions as input. It can be made more versatile if input in the form of a sum-of-product expression to the above described program can be implemented.

APPENDIX A. ORDERING FOR PLANAR BINARY DECISION DIAGRAM

The following tables, extracted from [6], list all threshold functions of 3 to 6 variables. It is found that **all** the threshold function have planar BDD. Many of the orderings are symmetrical due to identical weights. Only one of the symmetrical BDD is listed here. In the following table, in the ordering of 321 means $x_3x_2x_1$, with x_3 as the root node, and the x_1 as sink node in the BDD. For each threshold function in the table, a disjunctive form is expressed with the subscripts of the variables; for example, **1v23** denotes **x_1.OR.(x_2.AND.x_3)**.

The planar BDD for the following *three variable* threshold functions are found and tabulated with their orderings and the number of nodes, which indicates their compactness.

S/N	Weights; Threshold	Ordering	Number of nodes	Threshold Functions
1.	111;2	321	6	12v13v23
2.	211;3	321 132	5 5	12v13
3.	211;2	231 132	5 5	1v23

Table 1. Planar BDD for three variable threshold function .

S/N	Weights; Threshold	Ordering	Number of nodes	Threshold Functions
1.	2211;5	4321 2431 1234	6 6 6	123v124
2.	2211;2	4321 2431 2143	6 6 6	1v2v34
3.	1111;3	any order	8	123v124v134v234
4.	1111;2	any order	8	12v13v23v14v34v24
5.	2111;4	4321 1432	7 7	123v124v134

6.	2111;2	4321 1324	7 7	1v23v24v34
7.	2211;4	4321 2431 2134	7 8 7	12v134v234
8.	2211;3	4321 2431 2143	7 8 7	12v13v23v14v24
9.	3211;5	4321 2431 1432 1243	5 5 5 5	12v134
10.	3211;3	4321 2431 1432 1243	6 6 6 6	1v23v24
11.	3221;5	3421 3241 1432 1342	7 7 7 7	12v13v234
12.	3221;4	3421 3241 1432 1342	7 7 7 7	12v13v23v14
13.	2111;3	4321 1324	8 8	12v13v14v234
14.	3111;4	4321 1324	6 6	12v13v14
15.	3111;3	4321 1324	6 6	1v234

Table 2. Planar BDD for four variables threshold functions .

S/N	Weight; Threshold	Ordering	#nodes	Threshold function
1.	22211;2	54321 35421 32541 32154	7 7 7 7	1v2v3v45
2.	11111;4	any order	10	1234v1235v1245v1345v2345
3.	11111;2	any order	10	12v13v23v14v24v34v15v25v35v45
4.	22111;6	54321 15432 21543	8 8 8	1234v1235v1245
5.	22111;2	54321 25431 21543	8 8 8	1v2v34v35v45
6.	21111;5	54321 15432	9 9	1234v1235v1245v1345
7.	21111;2	54321 15432	9 9	1v23v24v34v25v35v45
8.	33211;8	54321 35421 25431 21543	7 7 7 7	123v1245
9	33211;3	54321 34521 25431 23541 21543 21354	7 7 7 7 7 7	1v2v34v35
10.	22211;6	54321 35421 23541 32154	9 10 10 9	123v1245v1345v2345
11.	22211;3	54321 35421 32541 32154	9 10 10 9	12v13v23v14v24v34v15v25v35

12.	32211;7	54321 35421 32541 13254	8 9 8 8	123v1245v1345
13.	32211;3	54321 35421 32541 15432 13452 13452	8 9 8 8 9 9	1v23v24v34v25v35
14.	33221;8	45321 43521 24531 12453	9 9 10 9	123v124v1345v2345
15.	33221;4	45321 43521 24531 21543 21354	9 9 10 9 9	12v13v23v14v24v34v15v 25
16.	11111;3	any order	11	123v124v134v234v125v1 35v235v145v245v345
17.	43221;9	45321 43521 25431 24531 14532 14352 12543 12453	8 8 8 8 8 8 8 8	123v124v1345
18.	43221;4	45321 43521 25431 24531 14532 12543 12453	8 8 8 8 8 8 8	1v23v24v34v25
19.	32221;7	43521 43251 15432 12543	9 9 9 9	123v124v134v2345
20.	32221;4	43521 43251 15432 14532	9 9 9 9	12v13v23v14v24v34v15

21.	33111;7	54321 25431 21543	7 7 7	123v124v125
22.	33111;3	54321 25431 21543	7 7 7	1v2v345
23.	22111;5	54321 25431 21543	10 11 10	123v124v125v1345v2345
24.	22111;3	54321 25431 21543	10 11 10	12v13v23v14v24v15v25v 345
25.	33221;7	45321 43521 24531 21543 21453	9 9 9 9 9	123v124v134v234v125
26.	33221;5	45321 43521 24531 21543 21453	9 9 9 9 9	12v13v23v14v24v345
27.	33222;7	54321 21543	10 10	123v124v134v234v125v 135v235v145v245
28.	33222;6	54321 21543	10 10	12v134v234v135v235v 145v245v345
29.	22211;5	54321 25431 32541 32154	9 10 10 9	123v124v134v234v125v 135v235
30.	22211;4	54321 25431 23541 32154	9 10 10 9	12v13v23v145v245v345
31.	32111;6	54321 25431 15432 12543	9 9 9 9	123v124v125v1345
32.	32111;3	54321 25431 15432 12543	9 9 9 9	1v23v24v25v345

33.	43221;8	43521 25431 24531 14532 14352 12543	10 10 10 10 10 10	123v124v134v125v2345
34.	43221;5	43521 25431 24531 14532 14352 12543	10 10 10 10 10 10	12v13v23v14v24v15v345
35.	43321;8	34521 32541 32451 15432 14532	10 9 9 9 9	123v124v125v135v134v 234
36.	43321;6	34521 32541 32451 15432 14532	10 9 9 9 9	12v13v14v23v234v345
37.	43322;8	32541 15432	11 11	123v124v125v134v135v1 45v234v235
38.	43322;7	32541 15432	11 11	12v13v145v234v235v245 v345
39.	33111;6	54321 25431 21543	8 10 8	12v1345v2345
40.	33111;4	54321 25431 21543	8 10 8	12v13v23v14v24v15v25
41.	22111;4	54321 25431 21543	9 12 9	12v134v234v135v235v14 5v245
42	33211;6	54321 35421 25431 23541 21543 21354	8 8 10 10 8 8	12v134v234v135v235

43.	33311;5	54321 35421 25431 23541 21543 21354	9 10 10 10 10 9	12v13v23v145v245
44.	53221;9	45321 43521 25431 24531 14532 14352 12543 12453	8 8 8 8 8 8 8 8	123v124v125v134
45.	53221;5	45321 43521 25431 24531 14532 14352 12543 12453	8 8 8 8 8 8 8 8	1v23v24v345
46.	32211;6	35421 32541 32451 15432 13542 12543	11 10 10 10 11 11	123v124v134v125v135v2 345
47.	32211;4	35421 32541 32451 15432 13542 12543	11 10 10 10 11 11	12v13v23v14v15v245v34 5
48.	32221;6	43521 43251 15432 14532	10 10 10 10	123v124v134v234v125v1 35v145
49.	32221;5	43521 43251 15432 14532	10 10 10 10	12v13v14v234v235v245v 345
50.	43111;7	54321 25431 12543	7 7 7	12v1345

51.	43111;4	54321 25431 12543	7 7 7	1v23v24v25
52.	54221;9	43521 25431 24531 43251 24351 15432 14532 14352 15243 12543	8 9 9 8 9 9 9 9 8 8	12v134v2345
53.	54221;6	43521 25431 24531 43251 24351 15432 14532 14352 15243 12543	8 9 9 8 9 9 9 9 8 8	12v13v14v15v23v24
54.	54321;9	34521 25431 24531 23541 23451 15432 14532 13542 13452 12543	8 9 9 9 9 9 9 9 9 8	12v134v135v234
55.	54321;7	34521 25431 24531 23541 23451 15432 14532 13542 13452 12543	8 9 9 9 9 9 9 9 9 8	12v13v14v23v245

56.	54322;9	25431 23541 15432 13542	11 11 11 11	12v134v145v135v235v23 5
57.	54322;8	25431 23541 15432 13542	11 11 11 11	12v13v145v234v245v235
58.	43311;7	35421 32541 15432 13542	8 8 8 8	12v13v234v235
59.	43311;6	35421 32541 15432 13542	8 8 8 8	12v13v23v145
60.	42211;7	54321 35421 54231 32541 15432 13542	8 9 8 8 8 9	123v124v134v125v135
61.	42211;4	54321 35421 54231 32541 15432 13542	8 9 8 8 8 9	1v23v245v345
62.	21111;4	54321 15432	11 11	123v124v134v125v135v1 45v2345
63.	21111;3	54321 15432	11 11	12v13v14v234v15v235v2 45v345
64.	43221;7	43521 25431 24531 24351 15432 14532 14352 12543	9 10 10 10 10 10 10 9	12v134v234v135v145

65.	43221;6	43521 25431 24531 24351 15432 14532 14352 12543	11 11 11 11 11 11 11 11	12v13v14v234v235v245
66.	32211;5	35421 32451 15432 13542	10 10 10 10	12v13v234v235v145
67.	31111;5	54321 15432	9 9	123v124v134v125v135v145
68.	31111;3	54321 15432	9 9	1v234v235v245v345
69.	53211;3	54321 35421 25431 23541 54312 35412 15432 13542 21543 12543 21354 21345	7 7 7 7 7 7 7 7 7 7 7 7	12v134v135
70.	53211;5	54321 35421 25431 23541 15432 13542 12543 12354	7 7 7 7 7 7 7 7	1v23v245
71.	32111;5	54321 25431 14532 12543	10 11 11 10	12v134v135v145v2345
72.	32111;4	54321 25431 14532 12543	11 11 11 11	12v13v14v234v15v235v245

73.	53311;8	35421 25431 35241 32541 23541 15432 13542 15423 15243 12543	8 8 8 8 8 8 8 8 8 8	12v13v2345
74.	53311;6	35421 25431 35241 32541 23541 15432 13542 15423 15243 12543	8 8 8 8 8 8 8 8 8 8	12v13v14v15v23
75.	53321;8	35421 34521 32541 32451 15432 14532 13542 13452	9 9 9 9 9 9 9 9	12v13v145v234
76.	53321;7	35421 34521 32541 32451 15432 14532 13542 13452	9 9 9 9 9 9 9 9	12v13v14v234v235
77.	42111;6	54321 25431 15432 12543	8 8 8 8	12v134v135v145
78.	42111;4	54321 25431 15432 12543	8 8 8 8	1v234v235v245

79.	42211;6	54321 35421 32541 15432 13542 13254	9 10 10 10 10 9	12v13v145v2345
80.	42211;5	54321 35421 32541 15432 13542 13254	9 10 10 10 10 9	12v13v14v234v15v235
81.	52211;7	54321 35421 32541 15432 13542 13254	7 7 7 7 7 7	12v13v145
82.	52211;5	54321 35421 32541 15432 13542 13254	7 7 7 7 7 7	1v234v235
83.	52221;7	45321 43521 42531 43251 15432	9 9 9 9 9	12v13v14v2345
84.	52221;6	45321 43521 42531 43251 15432	9 9 9 9 9	12v13v14v234v15
85.	31111;4	54321 15432	7 7	12v13v14v15v2345
86.	41111;5	54321 15432	7 7	12v13v14v15
87.	41111;4	54321 15432	7 7	1v2345

Table 3. Five variables threshold function.

S/N	Weight; Threshold	Ordering	#nodes	Threshold functions
1.	332221;7	546321 543621 216543 215643	13 13 13 13	123v124v134v125v135v145v126v234v235v245v3456
2.	222111;5	654321 365421 326541 321654	13 15 15 13	123v124v134v125v135v126v136v1456v234v235v236v2456v3456
3.	433221;8	326541 325641 156432 154632	13 13 13 13	123v124v134v125v135v145v126v136v234v235v2456v3456
4.	332111;6	654321 365421 265431 236541 216543 213654	11 11 11 11 11 11	12v134v135v136v1456v234v235v236v2456
5.	322211;6	436521 342651 165432 146532	11 13 13 15	123v124v134v125v135v145v126v136v146v234v2356v2456v3456
6.	543221;9	256431 254631 236541 235641 156432 136542 135642	14 14 14 14 14 14 14	12v134v135v145v136v234v235v2456
7.	433111;7	365421 326541 165432 136542 126543	12 12 12 12 12	12v13v1456v234v235v236

8.	432211;7	436521 265431 246531 243651 165432 146532 143652 126543	11 14 16 14 14 16 14 11	12v134v135v145v136v14 6v234v2356v2456
9.	321111;5	654321 265431 165432 126543	13 16 16 13	12v134v135v145v136v14 6v156v2345v2346
10.	533211;8	365421 245631 326541 324651 146532 136542 134652 165423	12 12 12 12 12 12 12 12	12v13v145v146v234v235 6
11.	422111;6	654321 365421 326541 165432 136542 132654	11 14 14 14 14 11	12v13v145v146v156v234 5v2346v2356
12.	522211;7	465321 346521 432651 165432 145632 143652	12 12 12 12 12 12	12v13v14v156v2345v234 6

Table 4. Planar BDD for 6 variables threshold functions

APPENDIX B. COMPACTNESS OF BDD FOR FIBONACCI FUNCTION

The following table illustrates the compactness of the BDD for fibonacci threshold function of 9,8,7,6,5 variables, which corresponds to column B,C,D,E,F in the table below. The profile of the number of nodes is symmetrical as the threshold varies from 1 to its maximum. The plot of theses nodes shows that the ratio of one gap to the next gap approximates the golden ratio. (Gap = distance from one minimal point to the next)

Threshold	B	C	D	E	F
1	11	10	9	8	7
2	11	10	9	8	7
3	11	10	9	8	7
4	13	12	11	10	9
5	11	10	9	8	7
6	13	12	11	10	9
7	13	12	11	10	9
8	11	10	9	8	7
9	15	14	13	12	9
10	13	12	11	10	7
11	13	12	11	10	7
12	15	14	13	12	7
13	11	10	9	8	
14	15	14	13	10	
15	15	14	13	10	
16	13	12	11	8	
17	15	14	13	10	
18	13	12	11	8	
19	15	14	13	8	
20	15	14	13	8	
21	11	10	9		
22	17	16	13		
23	15	14	11		
24	15	14	11		
25	17	16	13		
26	13	12	9		
27	15	14	11		
28	15	14	11		
29	13	12	9		
30	17	16	11		
31	15	14	9		
32	15	14	9		
33	17	16	9		

Table 1a. Number of nodes of fibonacci function at different thresholds. (To be continued in table 1b.)

34	11	10
35	17	14
36	17	14
37	15	12
38	17	14
39	15	12
40	17	14
41	17	14
42	13	10
43	17	14
44	15	12
45	15	12
46	17	14
47	13	10
48	17	12
49	17	12
50	15	10
51	17	12
52	15	10
53	17	10
54	17	10
55	11	
56	17	
57	15	
58	15	
59	17	
60	13	
61	15	
62	15	
63	13	
64	17	
65	15	
66	15	
67	17	
68	11	
69	15	
70	15	
71	13	
72	15	
73	13	
74	15	
75	15	
76	11	
77	15	
78	13	
79	13	

Table 1b . (Cont'd)

80	15
81	11
82	13
83	13
84	11
85	13
86	11
87	11
88	11

Table 1c . (Cont'd) Number of nodes of fibonacci function at different thresholds.

Figure 1. to Figure 5. show the plots of the profiles for the number of nodes versus the different thresholds of the fibonacci function.

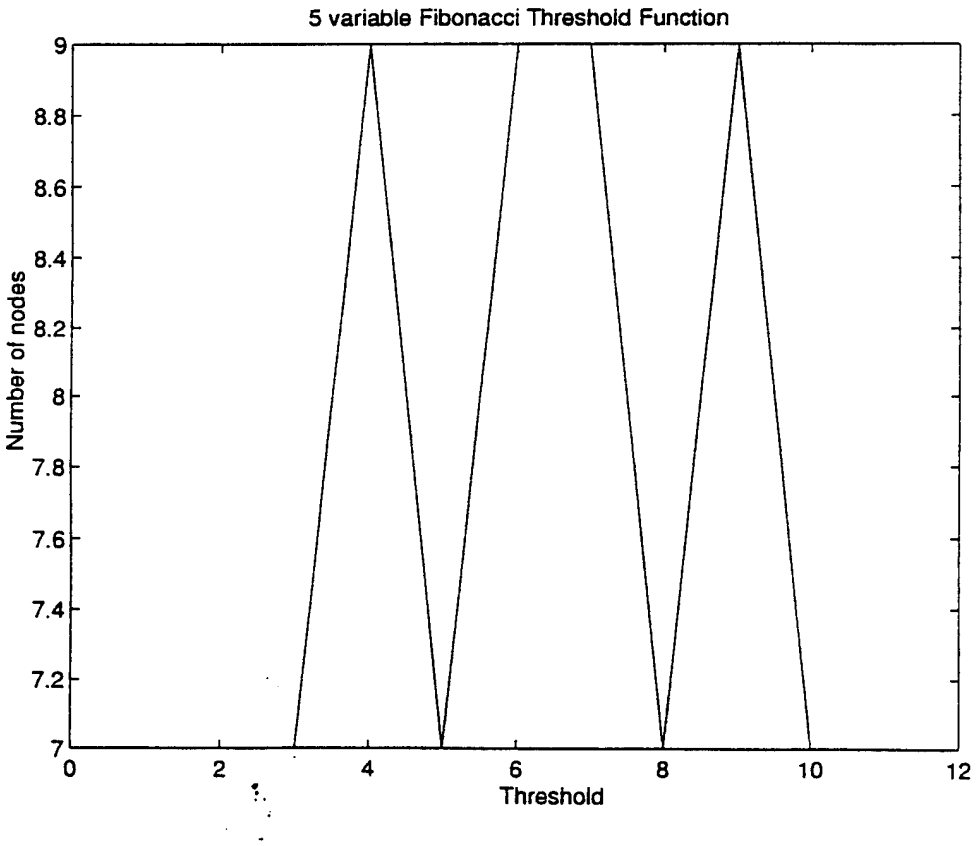


Figure 1. Compactness profile for F(5,3,2,1,1;T)

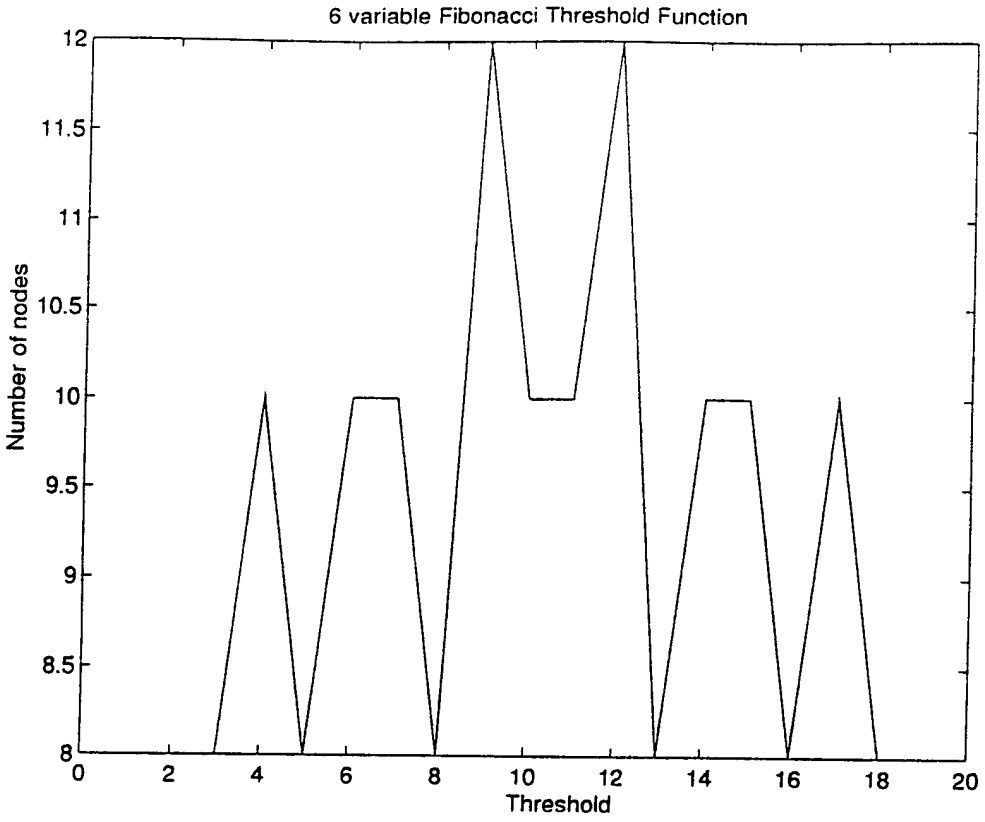


Figure 2. Compactness profile for F(8,5,3,2,1,1;T)

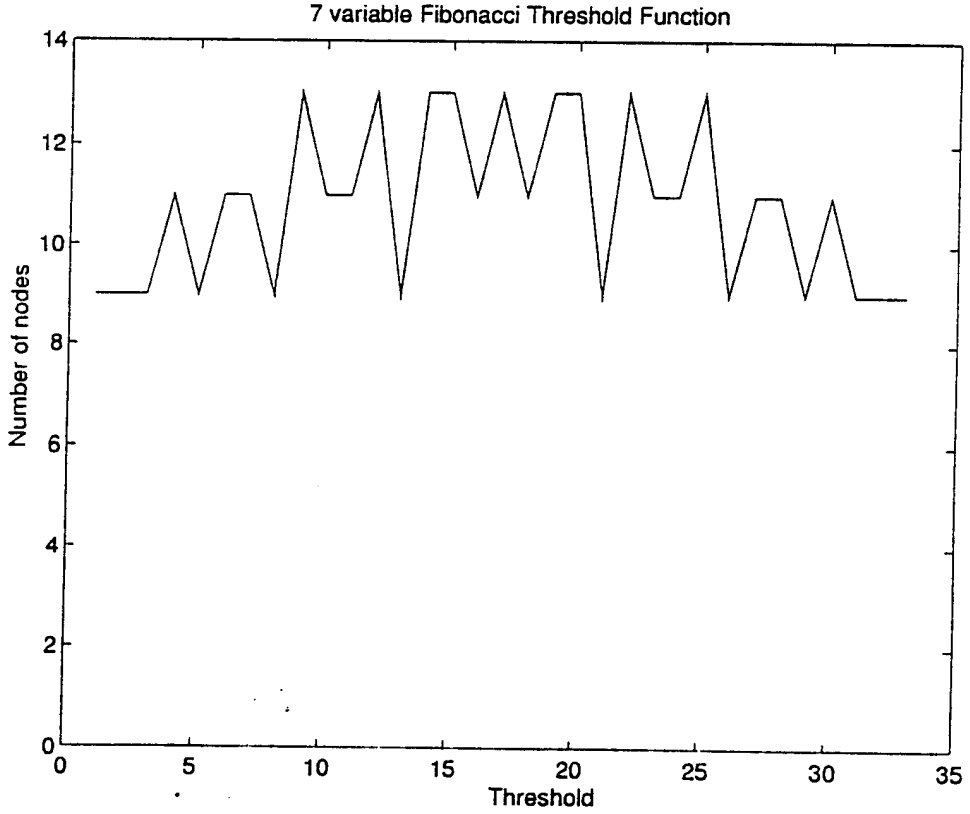


Figure 3. Compactness profile for F(13,8,5,3,2,1,1;T)

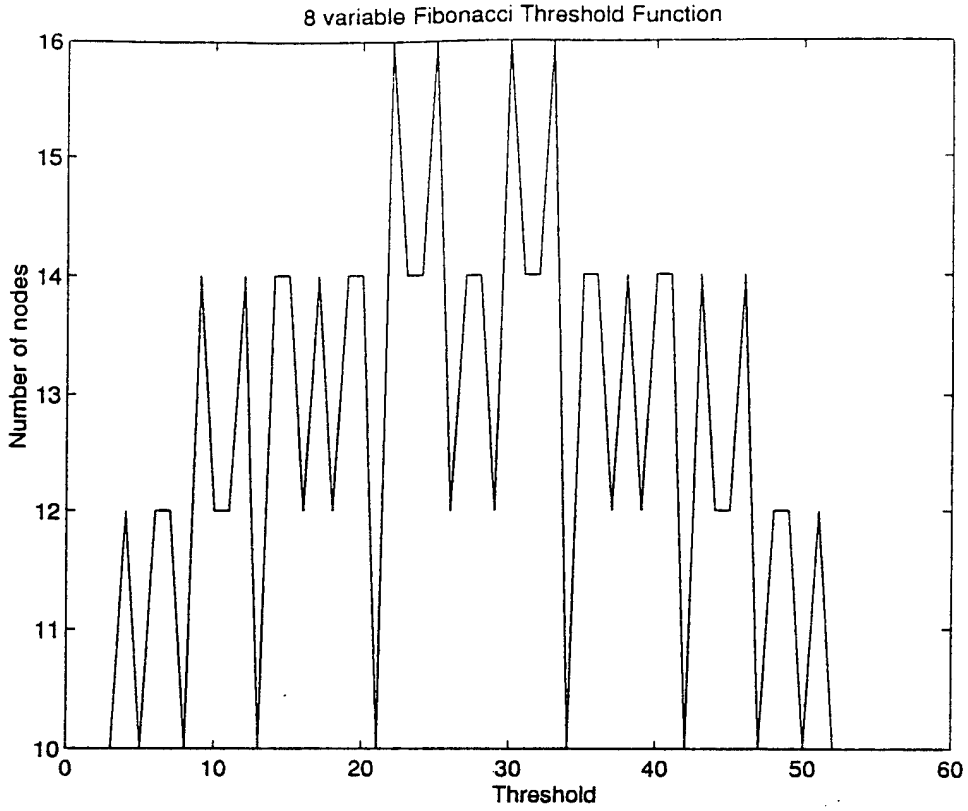


Figure 4. Compactness profile for F(21,13,8,5,3,2,1,1;T)

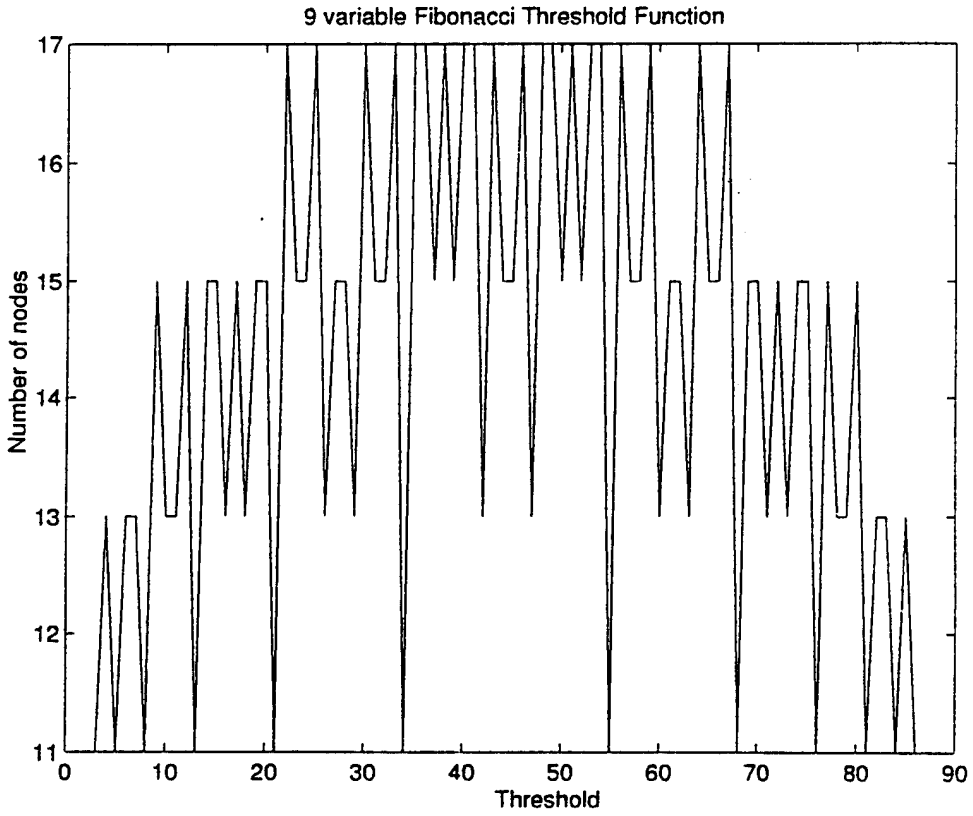


Figure 5. Compactness profile for F(34,21,13,8,5,3,2,1,1;T)

APPENDIX C. SOURCE CODE FOR BDD PROGRAM

```
/*************************************************************/
/*    Filename: TREE.H                                       */
/*                                                           */
/*************************************************************/


#define NULL 0
typedef char DATA;

struct node
{
     DATA d;
     int  index;
     struct node    *left;
     struct node    *right;
};

typedef   struct node    NODE;
typedef   NODE      *BTREE;


/********************** Prototypes ************************/

void print_to_file(int node_remain, DATA p[], BTREE root, int
num_var,  char  filename[],  int  threshold,  int  w[],  int
x_order[]);

BTREE init_node(DATA d1, int index, BTREE p1, BTREE p2);
BTREE new_node(void);
BTREE create_tree(DATA a[], int i, int size);
void inorder(BTREE root, int *cross_flag);
void preorder(BTREE root, int *node_remain);
void postorder(DATA p[],BTREE root,int *node_cnt, int num_var
);
DATA merge(BTREE node_ptr, int *node_remain);
void  print_tree(DATA  p[],  BTREE  root,  int  num_var,  int
x_order[]);
void cross_test(DATA d, int *cross_flag_ptr);
void merge_check( DATA *f_1[], int num_var);
DATA node(DATA node_loc[], BTREE root);
void data_set(int tf, int *threshold, int *w_in[]);
int sf1_cmp(DATA sf1[]);
```

```
/*************************************************************/
/* Filename: BDDANATL.C                                      */
/*                                                           */
/* BDD Tree Analysis Program for < 10 Variables              */
/*                                                           */
/* Functions:                                                */
/* a.Permutate all orderings of a Threshold Functions        */
/* b. Evaluate the Threshold Functions                       */
/* c. Generate BDD for each Function.                        */
/* d. Simplify the BDD.                                      */
/* e. Examine for Crossings.                                 */
/* d. Store results in ASCII file.                           */
/*   The results consist of                                  */
/*                                                           */
/*   i.   Ordering of variables that produce planar BDD      */
/*   ii.  Compactness of BDD in terms of no. of nodes        */
/*   iii. Binary Decision Diagram of the Function            */
/*************************************************************/


#include "tree.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

void main ()
{

int  min_node=1000,       /* Counter for searching for minimum
                             no. of nodes */
     carry[20],           /* Used in evaluation of threshold
                             function       */
     cmp_vector[20],      /* Vector used for comparison of
                             sub-functions    */
     found_flag,          /* Flag to indicate that a valid
                             ordering is found*/
     ii,                  /* Index */
     l,                   /* Index */
     m,                   /* Index */
     sf_size;             /* Size of subfunction to be compared
*/

char sf1[2050],           /*    String    describing    first
                             subfunction to be compared */
     sf2[2050];           /*    String    describing    second
                             subfunction to be compared */


int  t,                   /* Index */
```

```
        k,                  /* Index */
        i,                  /* Index */
        j,                  /* Index */
        f[2050],            /* Enumerated threshold function as
                               weighted sum */
        w_in[20],           /* Weights of threshold function
                               input */
        w[20],              /* Weights of threshold functions */
        node_remain=0,      /* Counter for number of nodes in
                               binary tree */
        *node_cnt,          /* Counter for number of nodes left
                               in binary tree */
        num_var,            /* Number of variable in a threshold
                               function */
        cross_flag=0,       /* Indicate that there is a crossing
                               if set */
        x_order[20],        /* Ordering of the weights in a
                               threshold function */
        threshold,          /* Threshold in a threshold function
                               */
        bin_wt[10],         /* Binary weights used for computing
                               leaf node address */
        bin_index[2050],    /* Leaf node address          */
        tmp_str[2050],      /* Temporary string used in BDD
                               manipulation */
        x[10][1024];        /* Max number of variable =10 */

DATA    f_vector[2050],     /* Vector describing threshold
                               function to be converted into binary
                               tree*/
        input_ok,           /* Input control flag    */
        filename[8],        /* Output filename  */

        f_1[2050],          /* Threshold function value in 1s &
                               0s,merging are marked here */
        f_lo[2050],         /* Unchanged Original Threshold
                               function vector */
        p[2050];            /* Data describing a node in binary
                               tree */


FILE        *ofp;           /* Pointer for output file */

BTREE       root;           /* Root address of binary tree */
```

```
/*------------ Data Input  -------------------*/

printf(" \n Number of variables : ");
scanf(" %d",&num_var);

for(i=0; i<num_var; i++)
{
printf("Weights :w%d=",i);
scanf("%d", &w_in[i]);
}

printf(" \n Enter threshold =");
scanf("%d",&threshold);

for(i=0; i<num_var; i++)
{
printf("Order  X:w%d=  ",i);
scanf("%d", &x_order[i]);
}

printf("Threshold Function : F( ");
printf(" Weight : ");
for ( i=0; i<num_var; i++)
printf("%d ",w_in[i]);
printf(" ; %d )",threshold);
printf("\n");

printf(" Order :");
for ( i=0; i<num_var; i++)
printf("%d ",x_order[i]);
printf("\n");

/* Loop for various  threshold ,provide the range of
thresholds that should be evaluated with the febonacci
functions here */

for (threshold=1; threshold<13; threshold++)
{

ofp=fopen("#node.5ve", "a");

fprintf(ofp,"Fibonacci function with %d variables  , and
threshold = %d ) \n",num_var,threshold );
```

```
/******Permutate for each threshold ************************/

for (i=0; i<20; i++)
     {
     carry[i]=0;
     if (i<num_var)
       x_order[i]=num_var-i;
     else
       x_order[i]=0;
     }
x_order[0]=num_var-1;

/**** generate increasing number for permutation generation */

do {
     x_order[0]++;
       for (i=0; i<num_var; i++)
       {
       x_order[i]=x_order[i]+carry[i];
       carry[i]=0;
            if (x_order[i]>num_var)
            {
            x_order[i]=1;
            carry[i+1]=1;
            }
       }

/**start: **** filter out the permutation ******/

for (j=1; j<=num_var; j++)     /* number to compare */
      {
     found_flag=0;
     for (i=0; i<num_var; i++)
       {
       if (x_order[i]==j)
            {
            found_flag=1;
            break;
            }
       }
      if (found_flag==0)       /*   if   found_flag=1   after
                               comparing
                               all elements of the array, it
                               is a valid permutation */
      break;
      }
```

```
 /********* filter out the permutation **********************/

 if (found_flag==1)
 {

 /* Here a permutation of an ordering is computated. The next
 thing to be done is to find out which of the Fibonacci
 function in this ordering are planar, and has the minimum
 number of nodes at the same time */

 min_node=1000;

 printf("\n\n");
 printf("\n  Threshold = %d  Ordering = ",threshold );
 for (j=0; j<num_var; j++)
    printf("%d",x_order[num_var-j-1]);
    printf("\n\n");

 /***********Permutation ends here **********************/


 for (i=0; i<num_var; i++)
   w[i]=w_in[(x_order[i]-1)];


 /*---- Initialisation ---------------------------------*/

 for (i=0; i<10; i++)
 bin_wt[num_var-i-1]=pow(2,i);

 for  (i=0; i<(pow(2,10)); i++)
 bin_index[i]=0;
 for (i=0; i<num_var; i++)
 {       for (j=0; j<pow(2,num_var); j++)
         x[i][j]=0;
 }

 for (i=0; i<num_var; i++)
 {       for (j=0; j<pow(2,num_var); j++)
         x[i][j]=0;
 }

 for (j=0; j<pow(2,num_var+1); j++)
         {
         f[j]=0;
         p[j]=' ';       /* initialise all nodes with the symbol
 blanks  */
         f_1[j]=EOF;
         f_1o[j]=EOF;
         }
```

```
p[2048]="\0";

/*-------Generate X0 X1 ... Xn  where n=num_var ----------*/

for (i=0; i<num_var; i++)
{
j=0;
do {
     for(k=0; k<(pow(2,i)); k++ )
     {
     x[i][j]=0;
     j=j+1;
     }

     for(k=0; k<(pow(2,i)); k++ )
     {
     x[i][j]=1;
     j=j+1;
     }

   } while ( j<pow(2,num_var) );        /* while loop */
} /* for loop */

/*--------- generate results of threshold function -------*/

for ( j=0; j<pow(2,num_var); j++)
     for (i=0; i< num_var; i++)
     {
     f[j]=w[i]*x[(num_var-x_order[i])][j]+f[j];

bin_index[j]=bin_wt[i]*x[(num_var-x_order[i])][j]+bin_index[
j];
     }

 for ( j=0; j<(pow(2,num_var)); j++)
 {
 if ((f[j] == threshold)||(f[j] > threshold))
       {

       f_1[(bin_index[j])]='1';
       f_lo[(bin_index[j])]='1';
       f[j]=1;
       }
 else
       {
       f_1[(bin_index[j])]='0';
       f_lo[(bin_index[j])]='0';
       f[j]=0;
       }
 }
```

```
/* Vector of threshold functions value is formed, the next
step is to simplify by comparing adjacent subfunctions.
Identical subfunction will be combined. */

/*--------Adjacent merge odd->even ---------------------- */

for (j=1; j<num_var; j++)
{

      for (i=0; i<(pow(2,num_var)); i++)
      {
      sf1[i]='\0';
      sf2[i]='\0';
      }

sf_size=pow(2,j);
      i=0;
      while (i<(pow(2,num_var)))
      {
        for (k=i; k<(sf_size+i); k++)
        {
        sf1[k-i] = f_lo[k];
        sf2[k-i] = f_lo[(k+sf_size)];
        }

if((strcmp(sf1,sf2)==0)&&(strcmp(sf1,"00")!=0)&&(strcmp(sf1,
"0000")!=0)&&(strcmp(sf1,"00000000")!=0)&&(strcmp(sf1,"00000
00000000000")!=0)&&(strcmp(sf1,"00000000000000000000000000000
0000")!=0)&&(strcmp(sf1,"00000000000000000000000000000000000
0000000000000000000000000000")!=0)&&(strcmp(sf1,"11")!=0)&&
(strcmp(sf1,"1111")!=0)&&(strcmp(sf1,"11111111")!=0)&&(strcm
p(sf1,"1111111111111111")!=0)&&(strcmp(sf1,"1111111111111111
1111111111111111")!=0)&&(strcmp(sf1,"11111111111111111111111
11111111111111111111111111111111111111111111")!=0))
        {
if((sf1!="00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000
000000000000000000")&&(sf1!="11111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111")
          {
if((sf1!="00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000
000000000000000000000000")&&(sf1!="11111111111111111111111
11111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111
            {
```

```
            for (m=(i+sf_size); m<(i+2*sf_size); m++)

            {
              if (sf_size==2)
                f_l[m]='a';
              else if (sf_size==4)
                f_l[m]='b';
              else if (sf_size==8)
                f_l[m]='c';
              else if (sf_size==16)
                f_l[m]='d';
              else if (sf_size==32)
                f_l[m]='e';
              else if (sf_size==64)
                f_l[m]='f';
              else if (sf_size==128)
                f_l[m]='g';
              else if (sf_size==256)
                f_l[m]='h';
              else if (sf_size==512)
                f_l[m]='i';
              else if (sf_size==1024)
                f_l[m]='j';
              }
          }
          }
        }
        i=i+2*sf_size;
      }
}
f_l[pow(2,num_var)]='\0';

/*-------- Adjacent merge even->odd ---------------------- */

for (j=1; j<num_var; j++)
{
for (i=0; i<(pow(2,num_var)); i++)
{     sf1[i]='\0';                    /* Initialisation   */
      sf2[i]='\0';
}
sf_size=pow(2,j);

      i=sf_size;
      while (i<(pow(2,num_var)))
      {
        for (k=i; k<(sf_size+i); k++)
        {
        sf1[k-i] = f_lo[k];
        sf2[k-i] = f_lo[(k+sf_size)];
        }
        /* l= sf1_cmp(sf1); */
```

```
if((strcmp(sf1,sf2)==0)&&(strcmp(sf1,"00")!=0)&&(strcmp(sf1,
"0000")!=0)&&(strcmp(sf1,"00000000")!=0)&&(strcmp(sf1,"00000
000000000000")!=0)&&(strcmp(sf1,"00000000000000000000000000000000
0000")!=0)&&(strcmp(sf1,"00000000000000000000000000000000000000
0000000000000000000000000000000")!=0)&&(strcmp(sf1,"11")!=0)&&
(strcmp(sf1,"1111")!=0)&&(strcmp(sf1,"11111111")!=0)&&(strcm
p(sf1,"1111111111111111")!=0)&&(strcmp(sf1,"11111111111111111
1111111111111111")!=0)&&(strcmp(sf1,"11111111111111111111111
11111111111111111111111111111111111111111")!=0))
       {

if((sf1!="00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
0000000000000000")&&(sf1!="11111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111")
          {

if((sf1!="00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000")&&(sf1!="11111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111")
           {
           for (m=(i+sf_size); m<(i+2*sf_size); m++)
           {
             l=i+sf_size+1;

if((sf_size==2)  &&  (f_l[l]!='b')  &&  (f_l[l]!='c')  &&
(f_l[l]!='d')  &&  (f_l[l]!='e')  &&  (f_l[l]!='f')  &&
(f_l[l]!='g') && (f_l[l]!='h'))
           f_l[m]='a';
else  if  ((sf_size==4)  &&(f_l[l]!='c')  &&(f_l[l]!='d')  &&
(f_l[l]!='e')  &&  (f_l[l]!='f')  &&(f_l[l]!='g')  &&
(f_l[l]!='h'))
           f_l[m]='b';
else  if ((sf_size==8) && (f_l[l]!='d') && (f_l[i]!='e') &&
(f_l[l]!='f') && (f_l[l]!='g') && (f_l[l]!='h'))
           f_l[m]='c';
else if ((sf_size==16) && (f_l[l]!='e') && (f_l[l]!='f') &&
(f_l[l]!='g') && (f_l[l]!='h'))
           f_l[m]='d';
else if ((sf_size==32) && (f_l[l]!='f') && (f_l[l]!='g') &&
(f_l[l]!='h') && (f_l[l]!='i') && (f_l[l]!='j'))
           f_l[m]='e';
else if ((sf_size==64) && (f_l[l]!='g') && (f_l[l]!='h') &&
```

```
(f_l[l]!='i') && (f_l[l]!='j'))
          f_l[m]='f';
else if ((sf_size==128) && (f_l[l]!='h') && (f_l[l]!='i')
&&(f_l[l]!='j'))
           f_l[m]='g';
else if((sf_size==256)&&(f_l[l]!='i')&(f_l[l]!='j'))
           f_l[m]='h';
else if ((sf_size==512)&&(f_lo[l]!='j'))
           f_l[m]='i';
else if (sf_size==1024)
           f_l[m]='j';
            }
            }
            }
        }
       i=i+2*sf_size;
     }
}
f_l[pow(2,num_var)]='\0';
```

```
 /*- Prepare result to be converted into binary tree -----*/

 for (i=0; i<(pow(2,num_var)-1); i++)
      f_vector[i]='*';

 for (i=(pow(2,num_var)-1); i<(pow(2,(num_var+1))+1); i++)
 {
      if (i==(pow(2,(num_var+1))))
      {
        f_vector[i]='\0';
        break;
      }
      else
      {
      if (f_l[i-(pow(2,num_var)-1)]=='0')      /* Lowest level
                                                subfunction */
        f_vector[i]='0';
      if (f_l[i-(pow(2,num_var)-1)]=='1')
        f_vector[i]='1';
      if (f_l[i-(pow(2,num_var)-1)]=='a')
        f_vector[i]='a';
      if (f_l[i-(pow(2,num_var)-1)]=='b')
        f_vector[i]='b';
      if (f_l[i-(pow(2,num_var)-1)]=='c')
        f_vector[i]='c';
      if (f_l[i-(pow(2,num_var)-1)]=='d')
        f_vector[i]='d';
      if (f_l[i-(pow(2,num_var)-1)]=='e')
        f_vector[i]='e';
      if (f_l[i-(pow(2,num_var)-1)]=='f')
        f_vector[i]='f';
      if (f_l[i-(pow(2,num_var)-1)]=='g')
        f_vector[i]='g';
      if (f_l[i-(pow(2,num_var)-1)]=='h')
        f_vector[i]='h';
      if (f_l[i-(pow(2,num_var)-1)]=='i')
        f_vector[i]='i';
      if (f_l[i-(pow(2,num_var)-1)]=='j')
      /* Highest level subfunction */
        f_vector[i]='j';
        }
}

/*----- Create Binary Tree-----------------------------*/

root=create_tree(f_vector,0,(pow(2,(num_var+1))-1)    );

for (i=0; i<num_var; i++)
preorder(root,&node_remain);              /* Call function to
                                          simplify binary tree
```

```
                                                    */

/* ----------        Output     -------------------------*/


cross_flag=0;
inorder(root, &cross_flag);        /* print out all nodes in
order & do crossing check */
node_remain=2;

*node_cnt=2;

/* Count the number of nodes & free memory used by binary
tree-- */

postorder(p, root, node_cnt, num_var);


if (cross_flag==2)
     {
       printf("\n Planar BDD\n ");
       for (j=0; j<num_var; j++)
           fprintf(ofp," %d",x_order[num_var-j-1]);

       fprintf(ofp,"  %d   %d\n ",threshold, *node_cnt);
       if (min_node>*node_cnt)
          min_node=*node_cnt; /* seek minimum nodes BDD */
       print_tree(p, root ,num_var, x_order);
     }
if (cross_flag==4)
    {
    if (min_node>*node_cnt)
     min_node=*node_cnt; /* seek minimum nodes BDD */
    printf("\nCrossing occurs ");
    }


/*------- Write to file if required----------------------*/

/* Ascending order */
     if (num_var==4)
     {
if((x_order[0]>x_order[1])&&(x_order[1]>x_order[2])
&&(x_order[2]>x_order[3]))
          print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
     }

     if (num_var==5)
     {
```

```
      i                                                              f
((x_order[0]>x_order[1])&&(x_order[1]>x_order[2])&&(x_order[
2]>x_order[3])&&(x_order[3]>x_order[4]))
           print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
      }

      if (num_var==6)
      {
      i                                                              f
((x_order[0]>x_order[1])&&(x_order[1]>x_order[2])&&(x_order[
2]>x_order[3])&&(x_order[3]>x_order[4])&&(x_order[4]>x_order
[5]))
           print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
      }

      if (num_var==7)
      {
      i                                                              f
((x_order[0]>x_order[1])&&(x_order[1]>x_order[2])&&(x_order[
2]>x_order[3])&&(x_order[3]>x_order[4])&&(x_order[4]>x_order
[5])&&(x_order[5]>x_order[6]))
           print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
      }


/* Descending order */

      if(num_var==4)
      {

if((x_order[0]<x_order[1])&&(x_order[1]<x_order[2])&&(x_orde
r[2]<x_order[3])&&(x_order[3]>x_order[4]))
           print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);

      }
      if (num_var==5)
      {

if((x_order[0]<x_order[1])&&(x_order[1]<x_order[2])&&(x_orde
r[2]<x_order[3])&&(x_order[3]<x_order[4]))
           print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
      }

      if(num_var==6)
      {
```

```
if((x_order[0]<x_order[1])&&(x_order[1]<x_order[2])&&(x_orde
r[2]<x_order[3])&&(x_order[3]<x_order[4])&&(x_order[4]<x_ord
er[5]))
          print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
     }

     if (num_var==7)
     {
     i                                                    f
((x_order[0]<x_order[1])&&(x_order[1]<x_order[2])&&(x_order[
2]<x_order[3])&&(x_order[3]<x_order[4])&&(x_order[4]<x_order
[5])&&(x_order[5]<x_order[6]))
          print_to_file(*node_cnt, p, root, num_var, filename,
threshold,w_in, x_order);
     }

} /* if a permutation is found */

     } while (carry[num_var]==0);      /* while permutation
is not completed */
fprintf(ofp, "\n Minimal BDD has %d  nodes \n",min_node);
fclose(ofp);

}    /* fibonacci Loop , from 1 to T_max */
     /* fclose(ofp); */

}    /* end of main */




/*-------------------------------------------------------*/

/*create a linked binary tree from an array */
/*****************************************/
BTREE create_tree(DATA a[], int i, int size)
{
if (i >= size)
     return NULL;
else
     return(init_node(a[i], i, create_tree(a, 2*i+1, size),
create_tree(a, 2*i+2, size)));
}
```

```
 /*************************/
 /* Creating a binary tree */
 /*************************/

 BTREE new_node()
 {
  return (malloc(sizeof(NODE)));
 }

 BTREE init_node(DATA d1, int i, BTREE p1, BTREE p2)
 {
  BTREE t;
  t=new_node();
  t->index=i+1;
  t->d=d1;
  t->left=p1;
  t->right=p2;
  return t;
 }

 /*-----Print out nodes in order of left to right--------*/

 void inorder(BTREE root, int *cross_flag_ptr)
 {
      if (root!=NULL)
      {
      inorder(root->left, cross_flag_ptr);
      cross_test(root->d,cross_flag_ptr);
      /* printf("%c", root ->d); */
      inorder(root->right,cross_flag_ptr);
      }
 }

 /*----- scan for mergeable nodes ------------------------*/

 void preorder(BTREE root, int *node_remain)
 {
      if (root!=NULL)
      {
      merge(root, &node_remain);
      preorder(root->left, &node_remain);
      preorder(root->right, &node_remain);
      }
 }

 /*-- count the number of nodes left in simplified BDD ----*/

 void postorder(DATA p[], BTREE root, int *node_cnt, int
 num_var)
 {
      DATA rd;
```

```
      int  i, discount=0;

      if (root!=NULL)

      {
      postorder(&p[0], root->left, node_cnt, num_var);
      postorder(&p[0], root->right, node_cnt, num_var);
      p[root->index]=root->d;
      rd=root->right->d;
      i=root->right->index;

      if ((root->d)=='*')
        *node_cnt = *node_cnt+1 ;

      if ((rd!='0')&&(rd!='1')&&(rd!='*'))
      {
        if((root->left->d=='0')||(root->left->d=='1'))
           discount=1;

if((i>(pow(2,(num_var-1))-1)&&(i<pow(2,num_var))&&(rd=='a')))
           discount=1;

if((i>(pow(2,(num_var-2))-1)&&(i<pow(2,num_var-1))&&(rd=='b'
)))
           discount=1;

if((i>(pow(2,(num_var-3))-1)&&(i<pow(2,num_var-2))&&(rd=='c'
)))
           discount=1;

if((i>(pow(2,(num_var-4))-1)&&(i<pow(2,num_var-3))&&(rd=='d'
)))
           discount=1;

if((i>(pow(2,(num_var-5))-1)&&(i<pow(2,num_var-4))&&(rd=='e'
)))
           discount=1;

if((i>(pow(2,(num_var-6))-1)&&(i<pow(2,num_var-5))&&(rd=='f'
)))
           discount=1;

if((i>(pow(2,(num_var-7))-1)&&(i<pow(2,num_var-6))&&(rd=='g'
)))
           discount=1;

if((i>(pow(2,(num_var-8))-1)&&(i<pow(2,num_var-7))&&(rd=='h'
)))
           discount=1;

if((i>(pow(2,(num_var-9))-1)&&(i<pow(2,num_var-8))&&(rd=='i'
```

```
)))
          discount=1;

if((i>(pow(2,(num_var-10))-1)&&(i<pow(2,num_var-9))&&(rd=='j
')))
          discount=1;
       if(discount==1)
            {
            *node_cnt=*node_cnt-1;
            discount=0;
            }
     }

     free(root);
     }
}



/************** Simplification routines for Binary Tree *****/

/* Return a '0/1' if left branch  & right branch  are both
'0/1' or else mark node with '*' to indicate that merging is
not possible */

DATA merge(BTREE node_ptr, int *node_remain_ptr)
{
DATA t,ln,rn;
ln=node_ptr->left->d;
rn=node_ptr->right->d;

if ((ln==rn)&&(node_ptr->index!=1))
   {
   if
((ln=='0')||(ln=='1')||(ln=='a')||(ln=='b')||(ln=='c')||(ln=
='d'))
     {
     t=node_ptr->left->d;
     node_ptr->d=t;
     node_ptr->left->d=' ';    /* terminate branch */
     node_ptr->right->d=' ';
     return t;
     }

     else if ((ln=='e') || (ln=='f') || (ln=='g') || (ln=='h')
|| (ln=='j'))

     {
     t=node_ptr->left->d;
     node_ptr->d=t;
     node_ptr->left->d=' ';    /* terminate branch */
```

```
      node_ptr->right->d=' ';
      return t;
      }

   }
}

/*----------------- Print result ---------------------*/

void print_tree(DATA p[], BTREE root, int num_var, int
x_order[])
{

int i,j;

printf("\n\n");
printf("X%d
%c",x_order[0],p[1]);

printf("\n\n");
printf("X%d                %c",x_order[1],p[2]);
printf("                                %c",p[3]);

printf("\n\n");
printf("X%d        %c",x_order[2],p[4]);
for (j=5; j<8; j++)
printf("               %c",p[j]);

if (num_var>2)
{

printf("\n\n");
     if (num_var!=3)
     printf("X%d    ",x_order[3]);
printf("%c",p[8]);
for (j=9; j<16; j++)
printf("       %c", p[j]);


if (num_var>3)
{
printf("\n\n");
       if (num_var!=4)
     printf("X%d  ",x_order[4]);

printf("%c",p[16]);
for (j=17; j<32; j++)
printf("   %c",p[j]);

if(num_var>4)
{
```

```
 printf("\n\n  ");
 for (j=32; j<64; j++)
 printf(" %c",p[j]);

 }
 }
 }
 }
 /*----------- Print result to file----------------------*/

 void print_to_file(int node_cnt, DATA p[], BTREE root, int
 num_var,
 char filename[], int threshold, int w_in[], int x_order[])
 {

 int i,j;
 FILE *ofp;

 ofp = fopen("D_perm.7vd","a");

 fprintf(ofp,"\nThreshold function  F=(");
 for ( i=0; i<num_var; i++)
 fprintf(ofp,"%d ",w_in[i]);
 fprintf(ofp,"; %d ) \n",threshold);

 fprintf(ofp,"Order    : ");
 for ( i=0; i<num_var; i++)
 fprintf(ofp,"X%d ",x_order[i]);
 fprintf(ofp,"\nNumber of nodes = %d \n\n",node_cnt);


 /******************* Literal ******************************/
  fprintf(ofp, "\Binary Tree Map\n");

 for (i=64; i<(pow(2,(num_var+1))) ; i++)
      {
      if (p[i]!=' ')
        fprintf( ofp, "Index = %d   %c  \n",i, p[i]);
      }
 fprintf(ofp, "\n\n");

 /************** Literal *********************************/


 /* ------------------------------level 1 */
 fprintf(ofp,"\n\n");
 fprintf(ofp,"
                %c", p[1]) ;

 /* ------------------------------level 2 */
```

```
fprintf(ofp,"\n\n");
fprintf(ofp,"                                    %c",p[2]);
fprintf(ofp,"
                %c",p[3]);

/* ------------------------------level 3 */

if (num_var>1)
{
fprintf(ofp,"\n\n");
fprintf(ofp,"                  %c", p[4]);
for (j=5; j<8; j++)
fprintf(ofp,"                                    %c", p[j]);
}

if (num_var==2)
fprintf(ofp,"\n\n");

/* ------------------------------level 4 */

if (num_var>2)
{
fprintf(ofp,"\n\n");
       /* if (num_var!=4)
       fprintf(ofp,"X%d       ",x_order[4]);   */
fprintf(ofp,"       %c ", p[8]);
for (j=9; j<16; j++)
fprintf(ofp,"                %c", p[j]);
}

if (num_var==3)
fprintf(ofp,"\n\n");

/* ------------------------------level 5 */

if (num_var>3)
{
fprintf(ofp,"\n\n");
fprintf(ofp,"   %c", p[16]);

for (j=17; j<32; j++)
fprintf(ofp,"       %c", p[j]);

}

if (num_var==4)
fprintf(ofp,"\n\n");

/* ------------------------------level 6 */

if (num_var>4)
```

```
 {

 fprintf(ofp,"\n\n");
 fprintf(ofp," %c",p[32]);

 for (j=33;j<64; j++)
 fprintf(ofp,"   %c",p[j]);

 }

 if (num_var==5)
 fprintf(ofp,"\n\n");

 /* ------------------------------level 7 */

 if(num_var>5)
 {
 fprintf(ofp,"\n\n");
 for (j=64; j<128; j++)
 fprintf(ofp,"%c ",p[j]);

 fprintf(ofp,"\n\n\n\n");
 }
 fclose(ofp);

 }


 /* ---------- Examing if there is crossing-------------*/

     cross_flag =0 :  no '0' or '1' occur yet
     cross_flag =1 :  first '0' occured
     cross_flag =2 :  first '1' occured
     cross_flag >=4 :  >1 '0->1' transition has occured

   ------------------------------------------------*/

 void cross_test(DATA d, int *cross_flag)

 {
 if ((*cross_flag==0)&&(d=='*'))
      *cross_flag=0;
 if ((*cross_flag==0)&&(d=='0'))
      *cross_flag=1;
 if ((*cross_flag==1)&&(d=='1'))      /* First crossing occurs
 all
 the time */
      *cross_flag=2;

 if ((*cross_flag==2)&&(d=='0'))
      *cross_flag=3;
```

```
if ((*cross_flag==3)&&(d=='1'))        /* > 1 crossing occurs
*/
     *cross_flag=4;
}


/*************************************************************

Compare sub-function to see that sf1 is not all '0' or all '1'
 before EOF;  return an integer 0 if the above is true, else
 return an integer 1  return -1 is error calling of function

*************************************************************/

int  sf1_cmp(DATA sf1[])
{
int  num_char=0,j=0;
int  cnt01=0;

while (sf1[j+1]!=EOF)
   {

if((sf1[j]==sf1[j+1])&&(sf1[j+1]!=EOF)&&((sf1[j]=='0')||(sf1
[j]=='1')))
     cnt01++;

   j++;
   }
while (sf1[num_char]!=EOF)
     num_char++;

return (num_char-cnt01);

 }   /* end of function */


/* ----- END ----- END ----- END ----- END ----- END ----- */
```

LIST OF REFERENCES

[1] J. T. Butler and T. Sasao, "Average Number of Nodes In Binary Decision Diagrams of Fibonacci Functions," preprint.

[2] R. E. Bryant, "Graph Based Algorithm for Boolean Function Manipulation.", *IEEE Transaction on Computer*, Vol C-35, No. 8, Aug 1986.

[3] T. Sasao and J. T. Butler, "A Design Method for Lookup Table Type EPGA by Pseudo-Kronecker Expansion," Mar 1994, preprint.

[4] T. Sasao and J. T. Butler, "Planar Multi-valued Decision Diagrams," Sept. 1994. Accepted. International Symposium on Multi-Valued Logic, May, 1995, Bloomington, IN.

[5] C. Y. Lee, " Representation of Switching functions by Binary Decision Diagrams," *Bell System. Technology Journal*, 38 (1959): 985-999.

[6] Muroga, *Threshold Logic and its Application*, Wiley Interscience, Appendix, 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4.	Jon T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
5.	David Herscovici, Code MA/Hc Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
6.	Ang, Kwee Hua RASF Block 253, No. 03-240 Bangkit Road Singapore 2367	1